UNIVERSITY OF OSLO
Department of Informatics

Assessing the Impact of Using Design Patterns of Enterprise Application Architecture

Master thesis

Artur Kamil Barczynski

January 31, 2014

Abstract

For successful businesses that develop software, software quality cannot be an exception - it must be the requirement. Design patterns are solutions to general problems that software developers face during software development. These solutions have generally been developed and evolved over time. It has been claimed that the design patterns have an impact on the quality of enterprise application architecture. In this paper we try to assess the impact of using design patterns of enterprise application architecture. We focused the scope of this thesis on complexity and maintainability. We chose several well-established measures for complexity and based on those measures we built three maintainability models. We chose four Web Presentation patterns as described by Martin Fowler in his work titled *Pattern of Enterprise Application Architecture*, namely: *a*) Model View Controller architectural pattern, *b*) Page Controller, *c*) Front Controller, and *d*) Template View.

For the purpose of this research we designed and developed four cases as a simple enterprise application. We based all four cases on the same requirements and the same technology but with a different set of Web Presentation patterns.

We compared the results after collecting the measures for complexity and maintainability for each case. The comparison shows that:

- the Model View Controller architectural pattern might have a positive impact on complexity and maintainability;
- using the Front Controller pattern instead of the Page Controller pattern in Model View Controller architecture might not have an impact on complexity and maintainability;
- when we use the Front Controller pattern instead of the Page Controller pattern in not Model View Controller architecture then the impact on complexity and maintainability might be positive.

This paper is not a complete evaluation of the impact of using design patterns of enterprise architecture, but it certainly is an introduction to design patterns and to measuring software qualities. We measured the results for a specific enterprise system by using a specific programming language, thus there is no generalization.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to express my great gratitude to Professor Eric Bartley Jul, my research Supervisor, for his patient guidance, enthusiastic encouragement and useful critiques of this research work.

I would also like to thank my family for their great support and patience which I received throughout my studies.

Finally, I wish to thank my fellow students and friends for their opinions whenever I was in doubt.

Chapter 1

Introduction

" Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."

Christopher Alexander

Software quality is considered to be one of the most significant concerns of software production teams. Additionally, design patterns are documented solutions for common design problems that are expected to enhance software quality. A review of the literature on the effects of applying a design pattern on software quality reveals that the results are controversial and that a safe conclusion cannot be made. This research work deals with collecting measures that can be used to measure the impact of using design pattern-based systems as enterprise applications. This work is based on static analysis of the source code for the case study application. We use the JHawk analysis tool to collect many complexity measures, such as: *a*) *Line of Code and Number of Statements*, *b*) *Halstead Complexity Measures*, *c*) *McCabe's Cyclomatic Complexity*, and *d*) *Object-Oriented Measures*. Based on the complexity measures we defined three maintainability models: *a*) *Omans's Maintainability Index*, *b*) *McCall's Maintainability model*, and *c*) *ISO/IEC 9126 model*.

1.1 Background

In October 21, 1994 Eric Gamma et al., in a book titled *Design Patterns: Elements of Reusable Object-Oriented Software*, presented recurring solutions to common problems in software design as a catalog of 23 software design patterns. These are considered as the standard of "good" software design. The authors claimed that using a particular software design solution, i.e design patterns, provides better maintainability and reusability. It is necessary to point out that the catalog created by Eric Gamma et al. is not the first and only catalog with design patterns. In the literature we can find other well-known patterns, e.g. *a*) game design patterns, *b*) computational patterns, *c*) patterns of enterprise application architecture, *d*) etc. Martin Fowler, in his work titled *Pattern of Enterprise Application Architecture*, cataloged design patterns that were closely related to enterprise application architecture. This book is divided into two parts: the first part is a short tutorial on enterprise application architecture, and the second part of the book is a reference to about 40 design patterns. The author describes how each pattern works and when to use it. He also provides code examples in a few popular programming languages. Many studies in the literature state that the design patterns offered by Eric Gamma and his colleagues improve the quality of object-oriented software systems. To sum up, design patterns aim to increase the quality of the system.

1.2 Motivation

Software development is an activity in which we write computer software. However, there are distinct kinds of software out there, each of which has its own challenges and complexities. In this thesis we concentrate on enterprise applications. Fowler [14] described the enterprise class of applications which often have complex data, i.e much of this complex data, to work on, together with the business rules that fail all tests of logical reasoning. There is no precise definition for an enterprise application, but we can give some indication of Fowler's meaning in Section 5.1. For some people the term "enterprise application" implies a large system. However, it is necessary to remember that not all enterprise applications are large, yet they can still provide much value to the enterprise. If a small project fails, it usually makes less damage than a large project. On the other hand, if we think about the cumulative effect of many small projects, then we can see a point in improving small projects. Improving several small projects can be very significant to an enterprise. It has been observed that a large number of enterprise applications are based on commonly occurring problems and that they differ only in the application of a specific code. Design patterns for enterprise applications means solving these problems possibly in the form of reusable code components. Patterns are formalized best practices that the programmer must implement oh his/her own in the application.

Enterprise applications provide business-oriented services such as online shopping and online-payment processing, online banking, customer relationship management, business intelligence, etc. There is a growing demand for enterprise applications that can handle these kinds services with high quality and efficiency. It is essential for an enterprise to improve the quality of an enterprise software system due to the cost of building and maintaining an enterprise application.

1.3 Goal

The goal of this thesis is to assess the impact of using design patterns of enterprise systems architecture. Design patterns have been advocated as a promising technique for achieving the reuse of software knowledge and improving software quality. This thesis aims at developing a quantitative approach to measuring the impact of using design patterns on maintainability and complexity in enterprise application architecture.

1.4 Approach

Our approach to assessing the impact of using design patterns on enterprise application architecture is to make a case study enterprise project and to develop a quantitative method for the measurement. For the purpose of this thesis we designed and implemented a simple enterprise application (E-Invoice). We used the case study enterprise project as a tool for the research. This research deals with collecting measures which we used to measure the impact of *Design Patterns* on complexity and maintainability. We would like to emphasize that implementation of the case study enterprise application was not the primary object of this study. In our opinion the best way to study *Design Patterns* is to begin coding with them. We chose the Java Enterprise programming language as the programming language. The Java Enterprise language is an object-oriented language and we will be using an object-oriented software development approach.

1.5 Work Done

One small enterprise project was developed for the purpose of this experimental work. The project is called E-Invoice. The main idea of the project was to build an enterprise application that offers an online service for invoicing, i.e. one service that handles multiple companies with many users at once. The choice of this kind of application was made randomly. The goal for this experimental work was to build a fully working service as an E-Invoice service. Due to time limitations and many difficulties during the set-up phase we were not able to fulfill the goal. Instead, we implemented four cases based on the main idea of the E-Invoice project but with a minimal set of functionality. All four cases have the same set of functionality and are based on the same technology used in the implementation. The only difference between them is that each case includes a different set of design patterns to achieve the goal. We describe in detail the case study enterprise project in Chapter 5. Secondly, we chose measures for complexity, and based on those measures we built three maintainability models. The next phase of the research was to collect measures which we used to measure the impact on complexity and maintainability. Finally, we tried to compare and discuss the measurements.

1.6 Results

The results from the experiments for the four case studies in which we implemented different sets with *Web Presentation* patterns show that these patterns influence almost all of the measures used in the experiment.

For the complexity measures the result values show that:

- by using both patterns *Template View* pattern and *Page Controller* pattern together we can reduce almost all of the complexity measures. This means that we can reduce complexity;
- by using the *Page Controller* pattern instead of the *Front Controller* pattern together with the *Template View* pattern we can obtain a system where the complexity possibly stays at the same level;
- by using the *Page Controller* pattern instead of the *Front Controller* pattern without the *Template View* pattern we can increase almost all of the complexity measures. This means that we can increase complexity;
- by using both patterns *Template View* pattern and *Front Controller* pattern together we can reduce almost all of the complexity measures. This means that we can reduce complexity.

For the maintainability models the result values show that:

- by using both patterns *Template View* pattern and *Page Controller* pattern together we can increase the average Maintainability Index value by at least 21.60 percent. We can also decrease both McCall's maintainability and the ISO/IEC 9126 maintainability values by at least 32 percent. This means that we can reduce maintainability;
- by using the *Page Controller* pattern instead of the *Front Controller* pattern together with the *Template View* pattern we can obtain a system where the maintainability possibly stays at the same level;
- by using the *Page Controller* pattern instead of the *Front Controller* pattern without the *Template View* pattern we can decrease the average Maintainability Index value by at least 3.28 percent. We can also increase both McCall's maintainability and the ISO/IEC 9126 maintainability values by at least 8.7 percent. This means that we can increase maintainability;

- by using both patterns *Template View* pattern and *Front Controller* pattern together we can increase the average Maintainability Index value by at least 17.58 percent. We can also decrease the McCall's maintainability value by at least 21.74 percent and decrease the ISO/IEC 9126 maintainability value by at least 26.09 percent. This means that we can reduce maintainability.

1.7 Evaluation

After an evaluation of our results we concluded the following statements:

- Patterns are solutions to common design problems. You learn the patterns and then recognize when your task can be solved by one rather than having to come up with a solution on your own.

- *Web Presentation* patterns may have a positive impact on the complexity and maintainability of the enterprise systems.

- Each of the researched *Web Presentation* patterns influenced the complexity and maintainability to a different grade. Some of them may have a better impact on complexity and maintainability than others. They cannot be used blindly, so you have to figure out which *Web Presentation* pattern fits your architecture and problem the best every time.

In this experimental evaluation of the results we used the following evaluation criteria:

- Software complexity is a software measure that is claimed to indicate the quality of the code. In the literature we found more than fifty different complexity measures which also capture different types of complexity. We collected, among others the *Number of Line of Code*(NLOC), the *Number of Statement*(NOS), Halstead Complexity Measures, McCabe's Cyclomatic Complexity and some measures which are strictly connected to the Object-Oriented design. The complete list with all the measures used to measure the impact of design pattern-based systems for all code module levels of examination is presented in Appendix A.

- The other criteria that we used in this research evaluation was software maintainability. Maintainability can be measured in several different ways. We chose three maintainability models, namely:

  - Oman's Models as a maintainability Index (MI) is a composite measure that incorporates a number of traditional source code measures into a single number that indicates relative maintainability.
  - McCall's Model is one of the earliest models presented by Jim McCall and his colleagues [32]. We introduced this model in Appendix B.
  - ISO/IEC 9126 Model, the International standard ISO/IEC 9126-3 defines maintainability as a set of attributes that bear on the effort required to make specified modifications (which may include correction, improvements or adoptions of software to environmental changes and modifications in the requirements and functional specification).

1.8 Contribution

In this thesis we try to assess the impact of using design patterns of enterprise application architecture through the quantitative method. We will answer the following questions that interest us:

- Which measures would be useful at the implementation level to measure the software qualities of the enterprise system?
- How do the values of measures that indicate code complexity get influenced by the use of a design pattern-based system?
- How does maintainability get influenced by the use of design pattern-based systems in the implementation of an enterprise application?

In this master's thesis we implemented the case study project which is a Web-based enterprise application for invoicing. The idea of the project was to build an enterprise application which could offer an online service for invoicing, i.e. one service that could handle multiple companies with many users at once. The case study project is described in detail in Section 5.1. Due to a time and space limitation we decided to limit our research area to four *Web Presentation* patterns. Therefore, as a minor contribution, this research work makes a rudimentary attempt to deal with assessing the complexity and maintainability of some *Web Presentation* patterns.

We would like to emphasize several aspects of this thesis:

- in this research work we focus specifically on enterprise design pattern-based systems implemented in object-oriented programming language;
- we focus on *Web Presentation* patterns in the form of reusable code components;
- we also aim at assessing the impact of using design pattern-based systems on maintainability and complexity.

1.9 Conclusion

We learned that *Design Patterns* can provide a toolbox of solutions to common problems. Overall, we concluded that a using design pattern-based system in most cases can help to reduce complexity. Using different kinds of *Web Presentation* patterns in the enterprise systems can affect maintainability to different grade.

1.10 Thesis overview

This research document is divided into seven chapters:

- Chapter 1: Introduction. This chapter contains a short description of Martin Fowler's work *Pattern of Enterprise Application Architecture* and Eric Gamma et al's. work *Design Patterns: Elements of Reusable Object-Oriented Software*. A short paragraph provides a brief description of the approach to the study of *Design Patterns*, goals and the case study prototype project that was done for the purpose of this paper. In this chapter we present the motivation, contribution, results and evaluation of results.
- Chapter 2: A Survey of Web Presentation Patterns. We progress through the main concepts associated with *Web Presentation* patterns and how to describe and use them. We also present examples of implementation in the Java Enterprise.

- Chapter 3: Measuring Software Qualities. In this chapter we present the background knowledge of measurement and measure. We also study different types of scales and software quality models.
- Chapter 4: Lists of some of the collected measures. It shows how the values of these measures are influenced by the use of design pattern-based systems. These measures indicate software complexity based on the definition of complexity by other researchers. Also, it determines how maintainability can be measured by using these measures based on predefined quality models. In this chapter we also describe the tools used to collect measures and other, alternative measurement tools omitted in this research project.
- Chapter 5: Experimental Work. We describe the case study project. This chapter presents the functional requirements for the enterprise application and technology used to implement the project. As selection of *Design Patterns* and experimental methodology is included in this chapter.
- Chapter 6: Experimental Process. We present four case studies which we implement for the purpose of this thesis. We discuss the experimental results and present an analysis of the complexity and maintainability of the example systems in a different configuration of the *Design Patterns*.
- Chapter 7 Future Directions and Conclusions is devoted to some concluding remarks and points out possible future research directions.

Part I

Literature Review

Chapter 2

A Survey of Web Presentation Patterns

2.1 Introduction

In this chapter we try to present only a few design patterns of Web presentation as described in the literature [14]. First, we introduce the notation we use to describe the design patterns. Then we present the *Model View Controller* architectural pattern. The World Wide Web adopted the Model View Controller architecture in all major programming languages as well as in Java Enterprise. The second pattern under study is a *Page Controller*, which is quite simple to understand. The idea of *Page Controller* is very similar to static HTML Web pages. Then we move on to the next design pattern, namely to the *Front Controller*. *Front Controller* is a more complex version of a controller than the *Page Controller*. The *Front Controller* should be used when we have complex Web sites, e.g. those with many similar things which we need to do when handling a request. The last design pattern which we attempt to study is a *Template View*. This is a pattern used as a view mostly built as a HyperText Markup Language(HTML) with markers. The markers can be resolved into calls to gather dynamic information. Since the static part of the page acts as a template for the response, Folwer [14] calls this a *Template View*.

2.2 The Structure of The Patterns

A study of design patterns offers a wide range of patterns, but for the purpose of this thesis we explain only some of them, namely, the design patterns that we found to be useful in the case study project implementation. When we describe design patterns we should use some notation, and we need a consistent, uniform structure. We try to explain design patterns by using the following template, and we use a structure that is similar to Fowler's [14]:

Name here we create a vocabulary that allows designers to communicate in a uniform way;

Intent here we sum up the pattern in a sentence or two;

Sketch here we we present a visual representation of the pattern, mostly with a UML diagram, but not always;

How It Works describes the solution;

When to Use It describes the motivating problem for the pattern;

Example shows a possible implementation for the pattern. We use modified examples from the case study project.

2.3 Model View Controller Pattern

2.3.1 Intent

For this thesis we are centered our discussion around the architecture of the *Model View Controller* software pattern. Although it was originally developed for personal computing by Trygve Reenskaug in the late 1970s, it has been widely adapted as an architecture for World Wide Web applications in all major programming languages, and also in the Java Enterprise.

2.3.2 Sketch

The sketch for this pattern is presented in Figure 2.1.

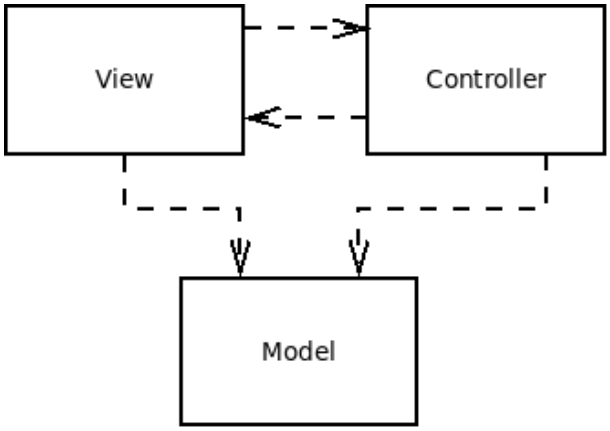

Figure 2.1: Model View Controller (MVC) pattern

Source: from ref. [14]

2.3.3 How It Works

The main purpose of the MVC pattern is the idea of separating business logic and the data access layer from the presentation layer. MVC consists of three elements. The Model consists of application data, business rules, logic, and functions. The View is a screen representation. Multiple views of the same data are possible, for example, the HTML Web page or PDF document of the same data. The Controller defines the way the user interface reacts to user input. This approach lets us attach multiple views to a model in order to provide different types of presentation. In addition to dividing the application into three kinds of components, the MVC design defines the interaction between them [15, 17].

- A Controller can send a command to the model to update the model's state(e.g. create or delete data). It can also send a command to its associated views to change the view's presentation of the model(e.g. next set of data to present).

- When a Model changes its state it then notifies its associated views and controllers. This approach makes it possible for views to produce update output and controllers to change the available set of commands. This approach is active implementation of the MVC pattern, but for the purpose of this thesis we used the passive implementation of the MVC pattern.

- A View requests from the model the data that it needs to generate an output representation. The view is only about display of information; any changes to the information are handled by the controller.

The most important point in this separation is the direction of the dependencies. The presentation layer depends on the model layer, but the model does not depend on the presentation layer. Why is this separation essential? Namely, the team programming in the model layer should be entirely unaware of what view as a presentation layer is being used; which for both sides simplifies their task and makes it easier to add a new presentation layer later on. If the team working with a presentation layer needs to make some changes to the code, the changes can be made freely without altering the model layer.

When we use this implementation then a common issue can arise; namely when we have a rich-client interface with multiple windows it is likely that there will be several presentations of a model on a screen at once. Now, if a user makes a change to the model from one presentation then the other will need to change as well. Fowler [14] proposes the following solution to do this. Without creating dependences we usually need an implementation of the *Observer pattern* [15], such as event propagation or a listener. With this solution the presentation layer acts as the observer of the model. Whenever the model changes, it sends out an event, and the presentation refreshes the information. For the scope of the case study project and this thesis we omit this active implementation and use the passive implementation of MVC instead.

In addition to the previous separation, where we separate the model and presentation layer, we also have separation of the view and controller. Why do we need a separate view and a controller? We can consider the following situation: when we need to support editable and non-editable behavior, which we can do with one view and two controllers for the two cases. In these cases the controllers are strategies [15] for the views.

2.3.4 When to Use It

Fowler [14] claims that the separation of presentation and model is one of the most powerful design principles in software, and the only time when we should not follow it is in very simple systems where the model has no real behavior in it. This separation between view and controller is claimed to be less significant than the previous one. Fowler recommends using it when it is truly helpful.

2.4 Page Controller

2.4.1 Intent

The idea of *Page Controller* is very similar to static HTML Web pages. In static HTML, Web pages are requested by sending the name and path for the static HTML document stored on the Web server. The approach of one path leading to one file that handles the request is a simple model to understand. The *Page Controller* has one input point to each page as a controller for each logical page of the Web site.The controller may be the page itself, as often is the case in a server page environment, or it may be a separate object that corresponds to the page [14].

2.4.2 Sketch

The sketch for the *Page Controller* pattern is presented in Figure 2.2.

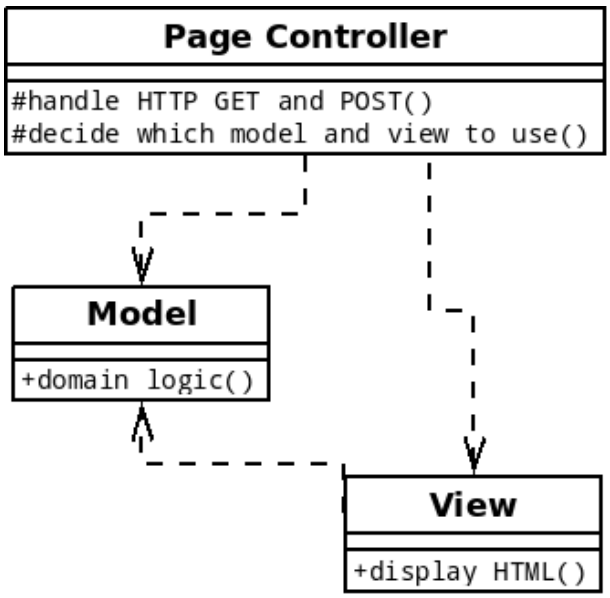

Figure 2.2: The Page Controller is an object that handles a request from a specific page or an action on a Web site.

Source: from ref. [14]

2.4.3 How It Works

The basic idea behind a *Page Controller* is to have one module on the Web server act as the controller for each page on the Web site. Scripting solutions such as PHP, ASP, JSP, etc. are based on this pattern. These scripting solutions provide a poor separation between the view and the controller when implementing a *Model View Controller* architecture. This can also provide difficulty during testing and performance. One way to a better architectural solution is to separate the view as a *Template View* from the other application logic. Fowler [14] claims that using a server page as a combination of a *Page Controller* and a *Template View* in one file works less efficiently for the *Page Controller*. The reason for this is that the *Page Controller* is more awkward to properly structure the module. If there is logic involved in either pulling data out of the request or deciding which actual view to display, then we can end up with an awkward scriptlet code in the server page. The primary responsibility of the *Page Controller* as part of the *Model View Controller* is to:

- decode the URL and extract data from the request;
- create and invoke any model objects to process the data. All of the data from the request should be passed to the model, so the model objects do not need any connection to the request;
- determine which view should be used to display model information on it.

2.4.4 When to Use It

When we are working with the MVC part of the controller then we can either use the *Page Controller* design pattern or the *Front Controller* design pattern as a part of the architectural MVC design pattern. Both of them have advantages and disadvantages. Fowler [14] claims that the *Page Controller*

- leads to a natural structuring mechanism where particular actions are handled by particular server pages or script classes; and
- works particularly well in a site where most of the controller logic is pretty simple.

In this thesis we assess the impact of using both the *Page Controller* and the *Front Controller*.

2.4.5 Example

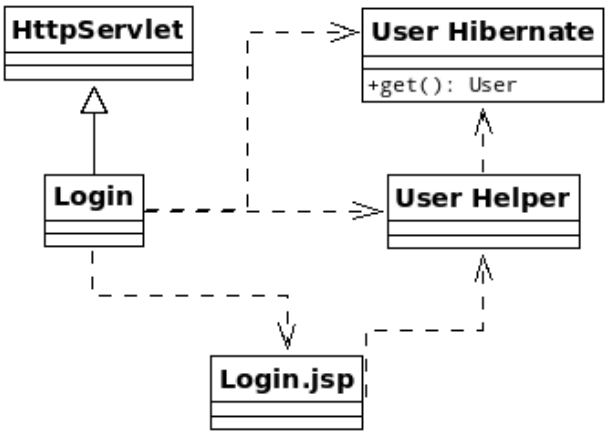

Figure 2.3: Classes involved in a Login page with a Page Controller servlet and a JSP view.

This is a striped example of a *Page Controller* Login page with an authentication service for the case study project. We display a page with forms to collect the username and the password, then we check the login credentials against those stored in the database. This service is available under the following URL: http://scorpius.arturkb.pl:8081/E-Invoice/user/login.do.

The Web server needs to be configured to recognize the /user/login.do as a call to the Login controller. In the listing, 2.1 is presented as a striped Tomcat configuration web.xml file.

Listing 2.1: Tomcat web.xml file

```
...
<servlet>
    <description>
    Login to the system</description>
    <display-name>Login</display-name>
    <servlet-name>Login</servlet-name>
    <servlet-class>pl.arturkb.EInvoice.Controller.User.Login</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>Login</servlet-name>
    <url-pattern>/user/login.do</url-pattern>
</servlet-mapping>
...
```

In listing 2.2 we implement a method to handle the request. Only the POST method is presented in the listing; other methods are omitted due to limited space and for simplicity's sake.

Listing 2.2: Login controller class

```

public class Login {
  ...

```

```
5   protected void doPost(HttpServletRequest request,
6     HttpServletResponse response)
7     throws ServletException, IOException {
8
9     Page page;
10  boolean loginSucess = false;
11
12  try {
13    // Begin unit of work
14    HibernateUtil.getSessionFactory().getCurrentSession().beginTransaction();
15
16    // Getting user
17    User user = (User) HibernateUtil.getSessionFactory().getCurrentSession().get(
      User.class, request.getParameter("username"));
18    if (user == null) {
19      loginSucess = false;
20    } else if (!user.getPassword().equals(request.getParameter("password"))) {
21      loginSucess = false;
22    } else if (user.getPassword().equals(request.getParameter("password"))) {
23      loginSucess = true;
24    }
25
26    HibernateUtil.getSessionFactory().getCurrentSession().getTransaction().commit()
      ;
27    // End unit of work
28
29    // Now we decide what to do, after checking loginSucess variable
30    if (loginSucess) {
31      HttpSession session = request.getSession(true);
32      user.setAuth(true);
33      session.setAttribute("user", user);
34      response.sendRedirect("/E-Invoice/dashboard/index.sec");
35      } else {
36        // Dispatcher
37        RequestDispatcher dispatcher = request.getRequestDispatcher("/E-Invoice/
      user/loginError.do");
38        dispatcher.forward(request, response);
39      }
40
41    } catch (Exception ex) {
42      HibernateUtil.getSessionFactory().getCurrentSession().getTransaction().
      rollback();
43      if (ServletException.class.isInstance(ex)) {
44        throw (ServletException) ex;
45      } else {
46        throw new ServletException(ex);
47      }
48    }
49  }
50  ...
51 }
```

In this case first the controller needs to create necessary model objects to do their thing, i.e. here it is just finding the correct model as a User object. Depending on the result from the model object, it puts the correct information in the HTTP response so that the controller can redirect to another. In this case, it creates a helper and puts it into the session. If the login credentials give a false result, then it forwards to the *Template View* to handle the display. The main reason for coupling between the *Template View* and the *Page Controller* are the parameter names in the request to pass on any objects that the JSP page needs. In a very similar way we can have behavior for Logout and the other components of the application. It is important to notice that the model should not include any servlet-dependent code, as any such servlet-dependent code

should be in a separate helper class. In this way we preserve the separation between the model and the presentation layer.

2.5 Front Controller Pattern (Mediator Pattern)

2.5.1 Intent

Fowler [14] states that when we have a complex Web site with many similar things which we need to do when handling a request, e.g. security, internationalization and providing particular views for a certain user, then the Controller part of the MVC pattern should be implemented as a *Front Controller*. It should consist only of a single servlet object as a handler which provides a centralized entry point for all the requests. This object can carry out common behavior which can be modified at run time with decorators. The handler then dispatches to the command objects for behavior that is particular to a request [14].

2.5.2 Sketch

The sketch for the *Front Controller* pattern is presented in Figure 2.4.

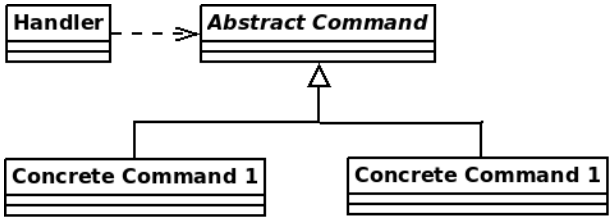

Figure 2.4: A controller that handles all requests for a Web site.

Source: from ref. [14]

2.5.3 How It Works

The main idea behind the *Front Controller* pattern is that it handles all calls for a Web site and that it is usually structured in two parts:

- Web handler. This object receives post or gets requests from the Web server. It parses and pulls just enough information from the URL and request to decide what action to initiate and then delegate to a command to carry out the action;
- command hierarchy. When we want to implement the commands then we often think about the classes rather than the serve pages. In this approach the classes do not need any knowledge of the Web environment. Mostly, HTTP information is passed on to them by the Web handler.

Figure 2.5 presents the idea of how the *Front Controller* works.

The Web handler is quite a simple part of the code whose only role is to decide to which command to run. There are also two versions of the *Front Controller*. We can divide them based on how they run the command, i.e. either statically or dynamically; *a*) the static version involves parsing the URL and uses conditional logic; *b*) the dynamic version usually takes some part of the URL and uses dynamic instantiation to create a command class.

One of the advantages of the static version is that it uses explicit logic, and this involves compile error checking on the dispatch component. This approach also makes room for quite

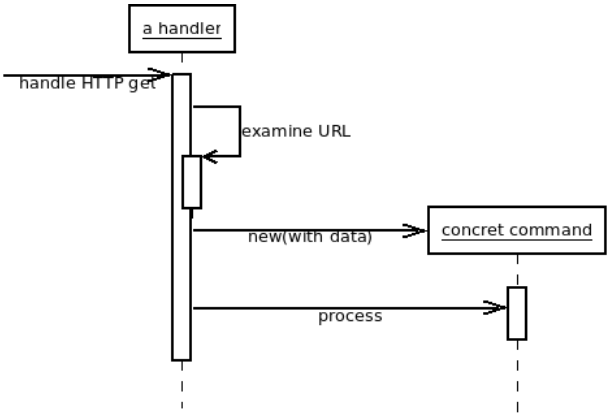

Figure 2.5: The *Front Controller* pattern approach

Source: from ref. [14]

a lot of flexibility with the URLs. We can statically map the URL to the command. On the other hand, this approach makes the process of adding new commands quite time consuming. The dynamic case allows to add a new command without changing the Web handler. One of the possible approaches with dynamic invocation is that we can put the name of the command class into the URL. Otherwise, we can use a properties file that binds the URLs to command the class name. The properties file is another file to edit, but it does make it easier to change class names without much searching through the Web pages. Flowler [14] claims that a particularly useful pattern to use with in conjunction with the *Front Controller* is *Intercepting Filter* [1]. This is essentially a decorator that warps the handler of the front controller, thus allowing us to build a *filter chain* to handle issues such as authentication, logging, and locale identification. We use filters to authenticate users, logging and internationalization in the case study project.

2.5.4 When to Use It

In the literature [1, 14] it is proposed to use the *Front Controller* in situations where the amount of controllers is quite large. With this we can omit duplication of code included within every *Page Controller*. Only one *Front Controller* has to be configured into the Web server. The Web handler does the rest of the dispatching. This simplifies the configuration of the Web server. When we use the dynamic implementation of the *Front Controller* commands where objects are created with each request, so we do not have to worry about making the command class thread-safe. We have to make sure that we do not share any other objects, such as the model objects.

2.5.5 Example

This is a striped example of a *Front Controller* Login page with the same functionality as in Example 2.4.5. We display the page with forms to collect the username and password, then we check the login credentials against those stored in the database. This service is available under the following URL with dynamic commands http://scorpius.arturkb.pl:8081/E- Invoice/-frontController?command=Login. The command parameter tells the Web handler which command to use. Figure 2.6 presents a UML diagram with the classes that implement the *Front Controller* pattern.

The Web server needs to be configured to recognize /frontController?command=Login as a call to the Login controller. Listing, 2.3 presents striped Tomcat configuration web.xml file.

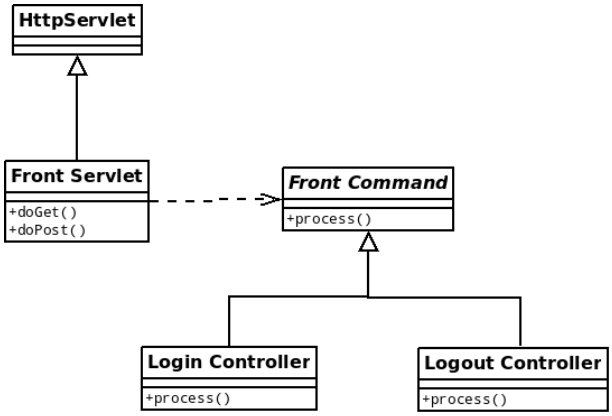

Figure 2.6: Classes that implement the *Front Controller* pattern approach

Listing 2.3: Tomcat web.xml file

```
...
<servlet>
    <description>
    This is controller version with Front Controller Design Pattern, for study
    purpose only
    </description>
    <display-name>FrontController</display-name>
    <servlet-name>FrontController</servlet-name>
    <servlet-class>pl.arturkb.EInvoice.FrontController.FrontServlet</servlet-class>
  </servlet>

<servlet-mapping>
    <servlet-name>FrontController</servlet-name>
    <url-pattern>/frontController</url-pattern>
  </servlet-mapping>
...
```

One of the first source codes which we want to present is an implementation of the Web handler. In listing 2.4 we implement a method to handle the Web request. Only the GET method is presented in the listing; other methods are omitted due to limited space and simplicity. The handler tries to instantiate a class named by connecting the command name and "Command". After the instantiate stage we move forward to initialize a command with the necessary information from the HTTP server. Then we call a "process" method on the wanted command. If the Web handler cannot find a command, we have used the *Special Case* pattern described by Fowler [14] which allows us to avoid much extra error checking.

Listing 2.4: Front Servlet class

```

public class FrontServlet extends HttpServlet {

  ...

  private static Logger logger = Logger.getLogger(FrontServlet.class);

  protected void doGet(HttpServletRequest request,
      HttpServletResponse response) throws ServletException, IOException {
    FrontCommand command;
    try {
      command = getCommand(request);
      command.init(getServletContext(), request, response);
      command.process();
```

```
15      } catch (Exception e) {
16        e.printStackTrace();
17      }
18    }
19
20    private FrontCommand getCommand(HttpServletRequest request) throws Exception {
21      try {
22        return (FrontCommand) getCommandClass(request).newInstance();
23      } catch (Exception e){
24        throw e;
25      }
26    }
27
28    private Class getCommandClass(HttpServletRequest request) {
29      Class result;
30        final String commandClassName = "pl.arturkb.EInvoice.FrontController." + (
        String) request.getParameter("command") + "Command";
31
32        try {
33          result = Class.forName(commandClassName);
34          logger.debug("Class: " + commandClassName);
35        } catch (ClassNotFoundException e) {
36          result = UnknowCommand.class;
37          logger.error("Class: " + commandClassName);
38        }
39      return result;
40    }
41
42    ...
43  }
```

The second source code which we want to present in listing 2.5 is an implementation of the Front Command. This class is abstract and the Commands share a fair bit of data and behavior. All of them need to be initialized with information from the Web server. The common behavior which they provide, as in this example, is forwarding. The process method is an abstract method which must be overridden by the extending class.

Listing 2.5: FrontCommand class

```
1
2  public abstract  class FrontCommand {
3    protected ServletContext context;
4    protected HttpServletRequest request;
5    protected HttpServletResponse response;
6
7    public void init(ServletContext context, HttpServletRequest request,
       HttpServletResponse response) {
8      this.context = context;
9      this.request = request;
10     this.response = response;
11   }
12
13   abstract public void process() throws ServletException, IOException;
14
15   protected void forward(String target) throws ServletException, IOException {
16     RequestDispatcher dispatcher = context.getRequestDispatcher(target);
17     dispatcher.forward(request, response);
18   }
19
20 }
21
22
23 }
```

The third source code for the *Front Controller* pattern that is presented in listing 2.6 is an implementation of the LoginCommand class. As in the examples of the source code above, we have omitted unimportant parts of the code due to simplicity's sake and to save space. The command class is very simple here. We override the process method when we get HashMap with a message translation for the application. Then we retrieve the proper message for the alert helper. We also use "makePage" method, not presented here to prepare "page" object and we put the information needed by the view into the request and forward it to a *Template View*.

Listing 2.6: LoginCommand class

```
...
public class LoginCommand extends FrontCommand {

  private static Logger logger = Logger.getLogger(Login.class);

  @Override
  public void process() throws ServletException, IOException {
    Page page;

    try {
      logger.debug("GET request for LOGIN");

      // Getting the language
      HashMap<String, String> lang = ServletsUtils.getLangMsg(request);

      // Alert preparation
      Alert alert = new InfoAlert();
      alert.setMsg(lang
          .get("Please_login_with_your_Username_and_Password."));

      // Page preparation
      page = makePage(request, alert);

      request.setAttribute("page", page);
      forward(page.getLayout());

    } catch (Exception ex) {
      if (ServletException.class.isInstance(ex)) {
        throw (ServletException) ex;
      } else {
        throw new ServletException(ex);
      }
    }
  }
  ...
}
```

The LogoutCommand class could be implemented in the same way.

The fourth and last source code for the *Front Controller* pattern that is presented in listing 2.7 is an implementation of the UnknowCommand class, which just brings up the error page for the application.

Listing 2.7: UnknowCommand class

```
public class UnknowCommand extends FrontCommand {

  public void process() throws ServletException, IOException {
    forward("/unknow.jsp");
  }

}
```

2.6 Template View

2.6.1 Intent

In this subsection we describe the *Template View* design pattern which is the view part of the *Model View Controller* architectural pattern. The presentation layer as a view is often built as a HyperText Markup Language (HTML) that is the main mark-up language for creating Web pages and other information that can be displayed in a Web browser. When we work with HTML pages as static pages, we mostly use tools that support WYSIWYG(What You See Is What You Get) editors. Most designers are quite comfortable with WYSIWYG editors. When we move on to dynamic Web pages, i.e. those where the context changes from request to request, then another problem occurs. In those pages we take the result of something such as database queries and embed them into the HTML. HTML editors cannot do the job with these dynamic pages. One of the solutions is what Fowler [14] presented: to compose dynamic pages as we do static pages but put in markers that can be resolved into a call to gather dynamic information. Since the static part of the page acts as a template for the particular response, Fowler calls this a *Template View* [14].

2.6.2 Sketch

The sketch for the *Template View* pattern is presented in Figure 2.7.

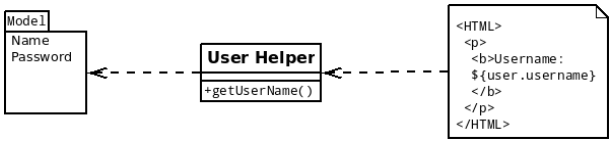

Figure 2.7: A view that processes domain data element by element and transforms it into HTML.

Source: from ref. [14]

2.6.3 How It Works

The basic idea of the *Template View* is to embed markers into a static HTML page when it is written. Then the mechanism works as follows: when the page is used to service a request, the markers are replaced by the results of some computation, for a example, database query. By followings this approach we can still prepare the page as an HTML page using WYSIWYG editors, and often by people who are not programmers. One of the most popular forms of the *Template View* is a server page such as JSP, PHP or ASP. In the case study projects we used a JSP server page. Fowler [14] presents a few road maps on how to use server pages. One of the first hints for a server page is that they allow to embed arbitrary programming logic, referred to as scriptlets, into the page. He claims that by putting many scriptlets into the page eliminates the possibility of nonprogrammers editing the page. He also claims that when we put a lot of scriptlets into the page we can easily wipe out the differences between layers of an enterprise application. So we should avoid scriptlets as much as possible. Another reason presented by Fowler for avoiding scriptlets is to provide a POJO[1] as a helper to each page. All real programming source codes should be written in the helper object, leaving the *Template View* page as clean as possible. The page only has calls into it which simplifies the page and

[1]POJO is an acronym for Plain Old Java Object. The name is used to emphasize that a given object is an ordinary Java Object, not a special object.

makes it purer than the *Template View*. In this way programmers can concentrate on helper POJOs and designers on the HTML part of the application [14].

2.6.4 When to Use It

When we work with the *Model View Controller* architectural pattern we can implement the view either as a *Template View* or as a *Transform View* [14]. Fowler [14] points to some strengths and weaknesses of the *Template View*. We try to summarize them in the short list below:

- he claims that one of the strengths of the *Template View* is that it allows us to compose the content of the page by looking at the page structure;
- he also points to two weaknesses, *a*) first, the common implementation makes it too easy to put complicated logic in the page, thus making it difficult to maintain; *b*) second, the *Template View* is harder to test than the *Transform View* [14]. Most of the implementations of the *Template View* are designed to work within a Web server and are very difficult or impossible to test otherwise.

2.6.5 Example

In this subsection we present an example of using a JSP as a view with a separate controller. For the controller part of this example we use an example of the *Page Controller* from listing 2.2, which is a Login service offered by the case study project. By using JSP as a view part of the MVC architecture it is important to pass to the JSP any information it will need to figure out what to display. One way to do this is to have the controller create a helper object and to pass it to the JSP by using the HTTP request. In this example we used a session to pass a helper object. When the controller is done with its own part of the work it passes the control to the server page which can now reach the helper by the useBean tag. We present the useBean tag in listing 2.8.

Listing 2.8: useBean tag

```
1 <jsp:useBean id="user" type "UserHelper" scope"session" />
```

Now with the helper object in place we can use it to access the information we need to display. In this example we used the *Data Mapper* pattern for the model which is described in [14]. The model information the helper needs was passed to it when it was created. We present the Helper class in listing 2.9. In this simple case, we provide a method to get and set the username.

Listing 2.9: User helper class

```
1  ...
2  public class User {
3   private String username;
4
5    public String getUsername() {
6    return username;
7    }
8
9    public void setUsername(String username) {
10   this.username = username;
11   }
12 ...
13 }
```

In listing 2.10 we present how we access this information by Java expression.

Listing 2.10: Example of using information as a Java expression

```
<div class="controls">
 <input type="email" id="inputUserName" placeholder=""
  value="${user.username}" readonly>
</div>
```

Fowler [14] proposes three ways of how to present the collection in the *Template View*. He claims that mixing Java and HTML is "horrible" and we agree with this statement. If we want to show a list of data, then we need to run a loop with a scriptlet in the server page. Another alternative is to move the loop to the helper class. The last alternative solution presented by Fowler [14] is by using a specialized tag for iteration. With this solution we keep the scriptlet out of the JSP and HTML out of the helper.

2.7 Chapter Summary

When we look at one of the biggest changes in enterprise applications in the last few years we can observe that the user Web browser-based interface has become more and more complex. The enterprise application offers many online services with no client software to install, a common UI approach and easy universal access. Having this observation in mind, we decided to study Web presentation patterns. Due to time limitations and the scope of this thesis we also decided to choose only the most relevant patterns for this research work and the case study project, namely:

- *Model View Controller*: We concluded that the main idea of the *Model View Controller* architectural pattern is to make a separation between business logic and the data access layer from the presentation layer. The most important point in this separation is the direction of the dependencies. The presentation layer depends on the model layer, but the model does not depend on the presentation layer. In addition to the previous separation, where we separate the model and the presentation layer, we also have a separation of view and controller.

- *Page Controller*: When we work with the controller part of the MVC then we can either use the *Page Controller* design pattern or the *Front Controller* design pattern as a part of the architectural MVC design pattern. Fowler [14] claims that the *Page Controller* *a*) leads to a natural structuring mechanism where particular actions are handled by particular server pages or script classes; *b*) works particularly well in a site where most of the controller logic is quite simple.

- *Front Controller*: As we mentioned above, one of the possibilities for the controller part can be *Front Controller* when to use the *Front Controller* as a controller for MVC architecture is motivated by the complexity of Web application. When the application becomes more and more complex we should consider using the *Front Controller* than *Page Controller*. There are also two versions of the *Front Controller*. We can divide them based on how they run the command, i.e. either statically or dynamically: *a*) the static version involves parsing the URL and using conditional logic; *b*) the dynamic version usually takes some part of the URL and uses dynamic instantiation to create a command class.

- *Template View*: The last design pattern which we decided to study is the *Template View*, which is the view part of the *Model View Controller* architectural pattern. Fowler [14] claims that one of the strengths of the *Template View* is that it allows us to compose the content of the page by looking at the page structure. Fowler [14] also points to two weaknesses: *a*) first, the common implementation makes it too easy to put complicated

logic into the page, thus making it hard to maintain; *b*) the second weakness is that the *Template View* is harder to test than the *Transform View* [14]. Most implementations of the *Template View* are designed to work within a Web server and are very difficult or impossible to test otherwise.

In the next chapter we will look at how to measure software qualities. Among other things we will describe why it is essential to measure software and to introduce concepts on which the measurement is based. At the end of the chapter we will present software quality models.

Chapter 3

Measuring Software Qualities

3.1 Background

Measurement is fundamental to any engineering discipline, and software engineering is one of these. We measure software for many different reasons:

- to indicate the quality of the product;
- to assess the productivity of the people who produce the product;
- to assess the benefits (in terms of productivity and quality) derived from new software engineering methods and tools;
- to form the baseline for estimation.

Lord Kelvin once said:

> "When you can measure what you are speaking about and express it in numbers, you know something about it; but when you cannot measure, when you cannot express it in numbers, your knowledge is of a meagre and unsatisfactory kind: it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the stage of a science."

Lords Kevin's words have been taken seriously by the software engineering community. It is undisputed that measurement is crucial for the progress of all sciences. Scientific progress is made through observation and generalization based on data and measurements, the derivation of theories as a result, and in turn the confirmation or refutation of theories via hypothesis testing based on further empirical data.

Before illustrating the various aspects of software measurement, we would like to explain that the term ”metric" has been often used instead of "measure" in the software measurement field in the past. As it has been pointed out by Sandro Morasca [33], "metric" has a more specialized meaning, i.e., distance, while "measure" is the general term. Therefore, we use "measure" in the remainder of this thesis.

3.2 Entities and Attributes

In this section we introduce two concepts on which measurement is based:

- *Entity*. An entity may be a physical object(e.g. a program), an event that occurs at a specified instant(e.g. milestone) or an action that spans over a time interval(e.g. the testing phase of a software project) [33];

- *Attribute*. An attribute is a characteristic or property of an entity(e.g. the size of a program, the time required during testing) [33]

When we want to measure an entity or attribute then we have to specify it first because measurement should not be used for entities or attributes alone. One of the best examples could be when we try to measure a program. This makes no sense since the attribute to be measured is not specified. In this example we could talk about size, complexity, maintainability, etc. Even if say to measure the size, since the entity whose size is to be measured is not specified, in this example it could be a specification, a program, a development team etc. Instead, we should measure the size of a program [33].

3.3 A Classification of Software Measure

Sandro Morasca [33] and Fenton and Pfleeger [11] also proposed to divide attributes into two categories: internal and external. We look at the definition of both categories.

- An internal attribute of an entity depends only on the entity, for instance, the size of the program may be considered an internal attribute since it depends only on the program.
- An external attribute of an entity depends on the entity and its context, for instance, the reliability of a program depends on both the program and the environment in which the program is used. It is important to measure the reliability of a program during its operational use so as to assess whether its quality is sufficiently high enough.

Sandro Morasca [33] also claims that external attributes are usually difficult to measure directly since they depend on the environment of the entity. On the other hand, internal products are easier to measure, but their measurement is seldom interesting *per se*, at least from a practical point of view. Internal product attributes are measured because they are believed to influence the external attributes(e.g. coupling is believed to influence maintainability) or the process (e.g. program complexity is believed to influence cost) [33].

Sandro Morasca [33] proposes to divide entities into two categories: product and process. He says that the product and process entities may be of a different kind.

- Product entities are any artifact produced or changed during software development and/or maintenance (e.g., source code, software design documents);
- Process entities are single phases, activities, and resources used during a project.

In this definition product and process entities have specific attributes, for instance, product attributes include size, complexity, cohesion, coupling or quality. When we describe process entities we can specify attributes such as time, effort, cost, etc.

3.4 Measurement and Measure

In this section we introduce a distinction between measurement and measure. This description is based on Sandro Morasca's definition.

- *Measurement*. Measurement is the process through which values(e.g. numbers) are assigned to the attributes of entities of the real world;
- *Measure*. A Measure is the result of the measurement process, so it is an assignment of a value to an entity with the goal of characterizing a specified attribute.

A measure defined as above is not just a value - it is a function that associates a value with an entity. The notation of value is by no means restricted to a real or integer number, i.e. a measure may associate a symbol with an entity; for example, a measure for the size of the program may associate the values "small", "medium" or "large" with the program.

3.5 Fundamentals of Measurements

We often begin measurement with the formulation of a theory or model. Then we must define and collect measures quantifying the key elements of the theory. An analysis of the collected data supports the evaluation of the accuracy of the original theory as well as the effectiveness of the measures themselves. Once a firm empirical foundation for a theory or model has been established, a practical application of the corresponding measures begins. Figure 3.1 illustrate the evolution of measures from theory to practice.

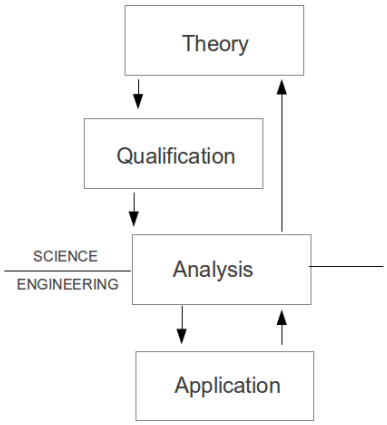

Figure 3.1: The Evaluation of Measures

Measurement is also used to overcome the intelligence barrier by converting an empirical observation into a numerical observation. These numerical observations are converted into numerical results by using some statistical and mathematical technique. The intent is for these numerical results to be easier to interpret than the original empirical observation. Once interpreted into empirically valid results we can use this information for whatever purpose we have in mind, subject to certain constraints. The generic measurement process is shown in Figure 3.2. It is obvious that a successful measurement activity should begin and proceed with clear objectives and goals.

3.6 Base vs. Derived Measures

When we categorize measures we will be using two categories for measures:

- Base measure: a measure that directly characterizes an empirical property and requires prior measurement of some other property;
- Derived measure: uses one or more base measures of one or more attributes to measure, indirectly, another supposedly related attribute. It requires first the measurement of two or more attributes, then it combines them bu using a mathematical model.

There is a relationship between internal and external attributes on the one hand and base and derived measures on the other. Internal attributes are often measured directly. Sometimes they must be measured using the values of the other internal measure, in which case they

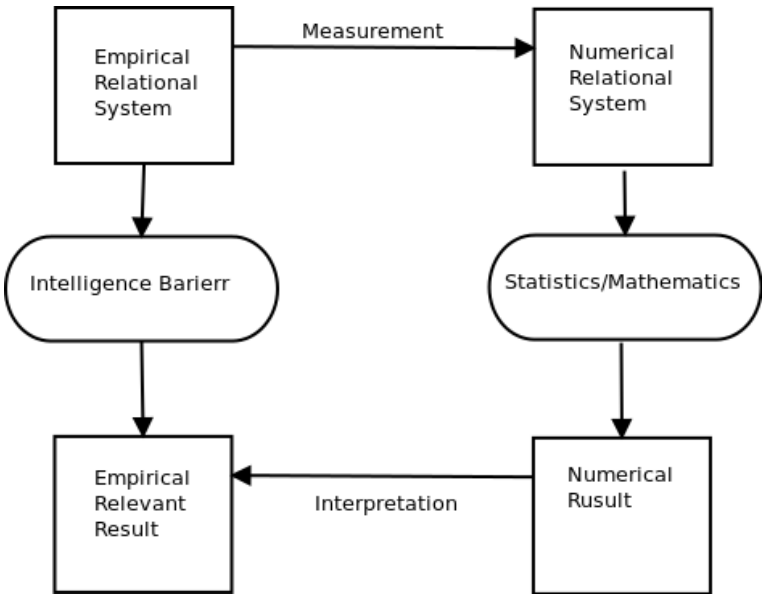

Figure 3.2: The Intelligence Barrier to Understanding

are called derived measures. External attributes are almost always measured with derived measures.

Examples of derived measures:

- Defect Density
- Reliability
- Productivity
- Maintainability

Examples of base measures:

- Cost
- Memory
- NLOC[Number of Lines of Code]
- Errors

3.7 Measurement Scale and Scale Types

To measure the quality of software inspection we may use a five-point scale to score the inspection, or we may use a percentage to indicate the inspection coverage. For some cases, more than one measurement scale is applicable. For others, the nature of the concept and resultant operational definition can be measured only with a certain scale. In this section we briefly discuss the four levels of measurement:[22]

- Nominal scale. Kan [22] describes this scale as a classification which is the most simple operation in science and the lowest level of measurement. In the classification process we attempt to sort elements into categories with respect to certain attributes. If we classify software products by the programming language that has been used, then we may have categories such as Java, C++, C, etc.;

- Ordinal Scale. Ordinal scale refers to measurement operations through which the subject can be compared in order, e.g. the degree of complexity of a module may be called low, medium or high. The ordinal measurement scale is at a higher level than the nominal scale in the measurement hierarchy;

- Interval Scale. An internal scale indicates the exact differences between measurement points. The mathematical operations of additions and subtractions can be applied to interval scale data; for instance, assuming products A, B and C are developed in the same programming language, if the defect rate of software product A is 5 defects per KLOC and product B's rate is 3.5 defects per KLOC then we can say that product A's defect level is 1.5 defects per KLOC higher than product B's defects level;

- Ratio Scale. The last level of measurement described by Kan [22] is the ratio scale. When an absolute or non-arbitrary zero point can be located on an interval scale, it becomes a ratio scale. All mathematical operations can be applied to it, including division and multiplication. Kan [22] also claims that measurement scales are hierarchical. Each higher-level scale possesses all the properties of the lower scales. The higher the level of measurement, the more powerful an analysis can be applied to the data. A higher-level measurement can always be reduced to a lower one, but not vice versa.

Kan [22] also claims that for the interval and ratio scale the measurement can be expressed in both integer and non-integer data. Integer data are usually given in terms of frequency counters(e.g. the numbers of defects the customer will encounter for a software product over a specified period of time).

Fenton and Pleeger [11] have one more scale in the hierarchy - the Absolute scale, which is described as the most restrictive of all. For measures M and M' there is only one admissible transformation - the identity transformation. In this case there is only one way in which the measurement can be made, so M and M' must be equal. The absolute scale has the following properties:

- Measurement for the absolute scale is made simply by counting the number of elements in the entity set;

- The attribute always takes the form of "number of occurrences of x in the entity";

- There is only one possible measurement mapping, namely the count;

- All arithmetic analyses of the resulting count are meaningful;

The uniqueness of the measures is an important difference between the ratio scale and the absolute scale [11]. Table 3.1 summarizes the key elements distinguishing the measurement-scale types that were discussed in this section.

The first two of these are often called "qualitative" scales, and the last three are called "quantitative" scales. We are in search of a quantitative measure for assessing the impact of using design pattern-based systems. Measures is a tool to help quantify the aspects of quality such that the effect of actions to improve quality can be measured.

3.8 Subjective and Objective Measurements

Sometimes a distinction is made between "objective" and "subjective" measures. The distinction is based on the way measures are defined and collected.

- *Objective*.

Table 3.1: Scales of measurements

Scale type	Admissible transformations (how measures M and M' must be related)	Examples
Nominal	$1-1$ mapping from M to M'	Labeling, classifying entities
Ordinal	Monotonic increasing function form M to M' that is, $M(x) \geq M(y)$ implies $M'(x) \geq M'(y)$	the degree of complexity of the module may called low, medium or high
Interval	$M = aM + b(a > 0)$	Relative time, temperature (Celsius, Fahrenheit) intelligence test (standardized scores)
Ratio	$M' = aM(a > 0)$	Time interval, length, temperature (Kelvin)
Absolute	$M = M'$	Counting entities

  - we define an objective measure in an unambiguous way;
  - usually the measurement process can be automated;
  - mostly there are no random measurement errors, i.e. the process is perfectly reliable;

- *Subjective*
  - leave some room for interpretation and require human intervention;
  - if we repeat the measurement of the same object(s) several times, we might not get exactly the same measured value every time, i.e. the measurement process is not perfectly reliable.

As a consequence, subjective measures are believed to be of lower quality than the objective ones. However, there are a number of cases in which objective measures cannot be collected, so subjective measures are the only way to collect pieces of information that may be important or useful.

Subjective measures usually entail well-defined measure procedures that precisely describe:

- how to collect data;
- how to conduct interviews;
- how to review documents;
- in which order to assess the dimension/items of the data collection instruments, etc.

Examples of well-defined measure procedures are:

- ISO900 Audit;
- CMMI/SPICE Assessment;
- Function Points

3.9 Software Quality Models

Kitchenham claims that quality is "hard to define, impossible to measure, easy to recognize" [23]. The presence of software quality is transparent from the users' point of view and that of professionals. The lack of software quality is easy to recognize. When we want to compare software quality in different situations, both qualitatively and quantitatively, it is necessary to establish a model for the software quality.

Yet another aspect of software quality is that of process quality versus end-product quality, from customer requirements to delivery of the software product. The development process is complex and often involves a series of stages, each with a feedback path. To improve quality during development, we need models of the development process, and within the process we need to select and deploy specific methods and approaches and employ proper tools and technologies. We need a measure of the characteristics and quality parameters of the development process and its stages as well as measures and models to ensure that the development process is under control and moving toward the product's quality objectives. The software end-product is the focus of this thesis, thus we will not study the quality of the software development process model in details.

There have been many models suggested for quality, most of these are hierarchical in nature. The quality models define software qualities as a hierarchy of factors, criteria and measures as is shown in Figure 3.3.

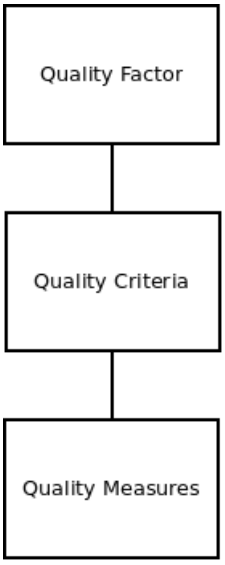

Figure 3.3: A Hierarchical Model of Quality

The *quality factor* represents the behavioral characteristics of the system. A *quality criterion* is an attribute of the quality factor that is related to software production and design. A *quality measure* is a measure that captures some aspects of the quality criteria. Two models which we are interested in being described are the following:

- McCall's Quality Model: One of the earliest software quality models was suggested by Jim McCall and colleagues [32]. As shown in Figure 3.3, the model defines software product qualities as a hierarchy of factors, criteria and measures. A quality factor represents a behavioral characteristic of the system. In Table 3.2 we list the quality factors as defined by McCall et al [32]. A quality criterion is an attribute of a quality factor that is related to software production and design. A quality measure is a measure that captures some aspect of a quality criterion [24]. In this model, eleven quality factors contribute to the definition of software quality. One or more quality measures should be associated

with each criterion. In Table 3.3 we list the 23 quality criteria defined by McCall et al. [32]

The relationship between quality factors and quality criteria is shown in Table 3.4. We can measure, for example, maintainability as a quality factor by combining consistency, simplicity, conciseness, self-descriptiveness and modularity. We should note that the relationships between the measures and the criteria that researchers seek to measure are usually quite complex. Another important point is that many aspects of quality can only be judged in relative terms; for example, it is difficult to imagine establishing an absolute reference point for usability. Usability can only be measured as relative to another experience.

- ISO 9126 Quality Model: This is an international standard for software quality measurement ISO 9126. The standards group has recommended six characteristics to form a basic set of independent quality characteristics. The quality characteristics and their definitions are shown in Table 3.5. The standard also includes a sample quality model that refines the features of IS0 9126 into several sub-characteristics, as Table 3.6 shows [24]. The standard provides a framework for organizations to define a quality model for a software product. On doing so, however, it leaves up to each organization the task of specifying precisely its own model. This may be done, for example, by specifying target values for quality measures which evaluate the degree of the presence of quality attributes.

Table 3.2: McCall's quality factors

Quality Characteristics	**Definition**
Correctness	The extent to which a program satisfies its specification and fulfills the user's mission objectives.
Reliability	The extent to which a program can be expected to perform its intended function with required precision.
Efficiency	The amount of computing resources and code that is required by a program to perform a function.
Integrity	The extent to which access to software or data by an unauthorized person can be controlled.
Usability	The effort required to learn, operate, prepare input and interpret the output of a program.
Maintainability	The effort required to locate and fix a defect in an operational program.
Testability	The effort required to test a program to ensure that it preforms its intended functions.

Continued on the next page

Table 3.2 – continued from the previous page

Quality Factors	Definitions
Flexibility	The effort required to modify an operational program.
Portability	The effort required to transfer a program from one hardware and /or software environment to another.
Reuseability	The extent to which parts of the software system can be reused in other applications.
Interoperability	The effort required to couple one system with another.

Source: from ref. [34]

Table 3.3: McCall's criteria contributing to software factors

Quality Criteria	Definition
Access audit	Ease with which software and data can be checked for compliance with standards or other requirements.
Access control	Provisions for control and protection of the software and data.
Accuracy	Precision of computations and output.
Communication commonality	Degree to which standard protocols and interfaces are used.
Completeness	Degree to which a full implementation of the required functionalities has been achieved.
Communicativeness	Ease with which inputs and outputs can be assimilated.
Conciseness	Compactness of the source code, in terms of lines of code.
Consistency	Use of uniform design and implementation techniques and notation throughout a project.

Continued on the next page

Table 3.3 – continued from the previous page

Quality Characteristics	Definition
Data commonality	Use of standard data representations.
Error tolerance	Degree to which continuity of operation is ensured under adverse conditions.
Execution efficiency	Run-time efficiency of the software.
Expandability	Degree to which storage requirements or software functions can be expanded.
Generality	Breadth of the potential application of software components.
Hardware independence	Degree to which the software is dependent on the underlying hardware.
Instrumentation	Degree to which the software provides for measurement of its use or identification of errors.
Modularity	Provision of highly independent modules.
Operability	Ease of operation of the software.
Self-documentation	Provision of inline documentation that explains implementation of the components.
Simplicity	Ease with which the software can be understood and tested.
Software system independence	Degree to which the software is independent of its software environment, non-standard language constructs, operating system, libraries, database management system, etc.
Software efficiency	Run time storage requirements of the software.
Traceability	Ability to link software components to the requirements.

Continued on the next page

Table 3.3 – continued from the previous page

Quality Characteristics	Definition
Training	Ease with which new users can use the system.

Source: from ref. [34]

3.10 Chapter Summary

Software quality is a complex concept. It means different things to different people, thus it is highly context-dependent. There is no universal definition of quality. In most cases, a single application is too complex and has to many aspects to be characterized from the measurements of single measure. In the next chapter we aim to answer the question which arises: "What software measures should we use?" One point of view is in conjunction with measuring the impact of using design pattern-based systems.

To assess the quality of the software we must measure certain important attributes, such as reliability, maintainability, and so on. These are external attributes and their measures are derived indirectly from other direct measures of attributes. In the next chapter we will consider the various important internal attributes which seem to play a role in measuring the external attributes. We will determine the collected measures and suggested indices for maintainability in order to find the impact of using design pattern-based systems in object-oriented programming.

Software quality criteria	Quality factor										
	Correctness	Reliability	Efficiency	Integrity	Usability	Maintainability	Testability	Flexibility	Portability	Reusability	Interoperability
Traceability	✓										
Completeness	✓										
Consistency	✓	✓				✓					
Accuracy		✓									
Error tolerance		✓									
Execution efficiency			✓								
Storage efficiency			✓								
Access control				✓							
Access audit				✓							
Operability					✓						
Training					✓						
Communicativeness					✓						
Simplicity		✓				✓	✓				
Conciseness						✓					
Instrumentation							✓				
Self-descriptiveness						✓	✓	✓	✓	✓	
Expandability								✓			
Generality								✓		✓	
Modularity						✓	✓	✓	✓	✓	✓
Software system independence									✓	✓	
Machine independence									✓	✓	
Communications commonality											✓
Data commonality											✓

Table 3.4: Relationship between McCall's quality factors and criteria.

Source: from ref. [34]

Table 3.5: ISO 9126 Quality Characteristics

Quality Characteristics	Definition
Functionality	A set of attributes that bear on the existence of a set of functions and their specified properties. The functions are those that satisfy stated or implied needs.
Reliability	A set of attributes that bear on the capability of software to maintain its performance level under the stated conditions for a stated period of time.
Efficiency	A set of attributes that bear on the relationship between software performance and the amount of resources used under the stated conditions.
Maintainability	A set of attributes that bear on the effort needed to make a specified modification (which may include correction, improvements or adoptions of software to environmental changes and changes in the requirements and functional specification).
Portability	A set of attributes that bear on the ability of software to be transferred from one environment to another(this includes the organizational, hardware or software environment).

Software quality measure	Quality characteristics: Functionality	Reliability	Usability	Efficiency	Maintainability	Portability
Suitability	✓					
Accuracy	✓					
Interoperability	✓					
Security	✓					
Maturity		✓				
Fault Tolerance		✓				
Recoverability		✓				
Understandability			✓			
Learnability			✓			
Operability			✓			
Time Behavior				✓		
Resource Behavior				✓		
Analyzability					✓	
Changeability					✓	
Stability					✓	
Testability					✓	
Adaptability						✓
Installability						✓
Conformance						✓
Replaceability						✓

Table 3.6: The IS0 9126 sample quality model refines the standards features into sub-characteristics.

Chapter 4

Measures Collected and Tools Used

Over the years it has been claimed that design patterns simplify the task of designing and building software. Eric Gamma et al. also claim that design patterns can improve the quality of software systems. This thesis aims at developing a quantitative approach to measuring the impact of using design patterns on complexity and maintainability in object-oriented programming. First, we introduce four different code module levels of examination. Then we present the software quality measures and models that can be used for measuring the impact of using design patterns in object-oriented programming. In the last part of this chapter we present the tools that are used in measurement as well some other, omitted, alternative tools.

4.1 Code Module Level of Examination

The case study project is written in Java Enterprise. Java Enterprise is an Object-Oriented programming language which offers a mechanism for organizing Java classes into namespaces. A Java package is usually used to organize classes belonging to the same category or providing a similar functionality. More about the case study project can be found in Section 5.1. Since we used Object-Oriented programming language for the case study project within quite a complex architecture, an approach as a simple, single file program is not realistic. Instead, we use different code module levels of examination for measurement. We use the following four levels of examination:

- *Method Level*. Measures at this level are measures that can give us some guidance as to the quality of code at the method level.
- *Class Level*. At the class level we look not just at the measures that measure aspects of the class, but we also look at measures which give us information on interaction between the classes. Measures which measure these class interactions tell us far more about the design than about the code. Some of the measures tell us how good our 'division of labor' is between the methods, while others tell us how much a change to a particular class will affect the code in an other class [27].
- *Package Level*. Measures at the package level;
- *System Level*. The system level is defined as all of the Java classes that we selected for analysis.

The relationship between measures and code module levels is presented in Table 4.1 and in Table 4.2. Measures are presented in Appendix A

Software complexity measure	Code module level: Method level	Class level	Package level	System level
NLOC	✓	✓	✓	✓
COMP	✓			
AVCC		✓	✓	✓
TCC		✓	✓	✓
NOMT		✓	✓	✓
NOCL	✓			
NOS	✓	✓	✓	✓
HLTH	✓		✓	✓
NAND	✓			
NOPR	✓			
HVOC	✓			
UAND	✓			
UOP	✓			
HEFF	✓	✓	✓	✓

Table 4.1: Relationship between measures and code module levels - part one.

4.2 Complexity

In this section we briefly present some of the software measures that claim to indicate the complexity for the code. In the literature we found more than fifty different complexity measures which also capture different types of complexity. Selection of measures is based on measures which are pre-implemented in the JHwak measures collecting tool. We have omitted some of them, namely those which, in our opinion, were not relevant to the research project. The full list with all of the measures used to measure the impact of design pattern-based systems for all code module levels of examination is presented in Appendix A.

4.2.1 Line of Code and Number of Statement

The number of *Lines of Code* (NLOC) is one of earliest and most widely used measures of size. The success of NLOC is due to the fact that it is easy to understand and to measure. When we examine the lines of code at the higher levels its usefulness is limited, but at the method level it is quite different. The big method can have a negative impact on the quality of software. If methods are long they can be difficult to understand. The high value of this measurement can also indicate that the method is trying to do too much. One of the rules of programming is that each method should perform a single, clear and distinct function. In our research evaluation we will use measure other than lines of code, namely the number of Java statements. The authors of JHawk tools claim that *Number of Statements* (NOS) is a more usable measure in Java language, but we still collect NLOC [29, 33].

Software complexity measures	Code module level: Method level	Class level	Package level	System level
HDIFF	✓			
HBUG	✓	✓	✓	✓
HVOL			✓	
CREF	✓			
XMET	✓			
LMET	✓			
UWCS		✓		
INST		✓	✓	
PACK		✓		
RFC		✓		
CBO		✓		✓
CCML		✓	✓	✓
DIST				✓
CCOM				✓
FIN		✓	✓	✓
FOUT		✓	✓	✓
LCOM		✓		✓
LCOM2		✓		
EXT		✓		

Table 4.2: Relationship between measures and code module levels - part two.

4.2.2 Halstead Complexity Measures

The Halstead Complexity measures were introduced by Maurice Howard Halstead in 1977 [18] as part of his treatise on establishing an empirical science of software development. Halstead claimed that the measures of software should reflect the implementation or expression of algorithms in different languages, but that they should be independent of their execution on a specific platform. These measures are static analysis of the program source.

All of the measures presented below are relevant at the method level and can reasonably be presented as averages per method at the class, package and system level: [29]

- *Halstead Length* (HLTH). This is a measure which simply sums up the numbers of operators and operands - a small number of statements with a high Halstead Volume would suggest that the individual statements are quite complex.
- *Halstead Vocabulary* (HVOC). This measure gives a sense of the complexity that exists among the statements, e.g. whether one is using a small number of variables repeatedly

(less complex) or one is using a large number of different variables, which will inevitably be more complex.

- *Halstead Difficulty* (HDIF) A measure which uses a formula to assess the complexity based on the numbers of unique operators and operands. It suggests how difficult the code is to write and maintain.
- *Halstead Effort* (HEFF). Attempts to estimate the amount of work it would take to recode a particular method.
- *Halstead Bugs* (HBUG). Attempts to estimate the number of bugs that are liable to be in a particular piece of code.

4.2.3 McCabe's Cyclomatic Complexity

In 1976 McCabe [30] developed a system which he called the cyclomatic complexity of a program. This is a measure of the number of possible alternative paths through a piece of code. At the method level, when one line of code simply follows another, we can obtain a low cyclomatic complexity value. On the other hand, methods with a high cyclomatic complexity value are those which have many if, for and while statements, loops, etc. Cyclomatic complexity values over 10 are generally viewed as bad [29].

4.2.4 Object-Oriented Measures

In this subsection we will present only some of the measures which are strictly connected to the Object-Oriented design. The measures that have been most widely discussed and accepted are those defined in [6], which we will now concisely describe.

- The first of them is Lack of Cohesion in Methods(LCOM) - LCOM measures the correlation between the methods and the local instance variables of a class. When the cohesion value is high, it indicates good class subdivision; otherwise, we should look into subdividing the classes into two or more subclasses. The lack of cohesion or a low cohesion value increases complexity. In this study we use the LCOM and LCOM2 Henderson-Sellars [19] version of Lack of Cohesion.
- One measure of Coupling is *Response For Class* (RFC). This measures the complexity of the class in terms of method calls. It is calculated by adding the number of methods in the class (not including inherited methods) plus the number of distinct method calls made by the methods in the class (each method call is counted only once, even if it is called from different methods) [29].
- *Coupling between Objects* (COB) - this measure is a count of the number of classes that are coupled to a particular class, i.e. where the methods of one class call the methods or access the variables of the other. These calls need to be counted in both directions that the CBO of class A is the size of the set of classes that class A references and those classes that reference class A. Since this is a set, each class is counted only once even if the reference operates in both directions, i.e. if A references B and B references A, B is only counted once. Chidamber and Kemerer claim that CBO should have as low a value as possible.

4.3 Maintainability

In this section we present three different models to compute the maintainability indices. There are many models for quantifying the software maintainability from software measures that

have been defined and implemented in past years. The choice for those three models is based on the fact that all of them are well documented and many studies have been carried out on them. For the purpose of this study we need to implement some of the measures in the JHawk measures collecting tool. A full list with all the measures used in the three models described is in Appendix B. The three models for object-oriented programming are as follows:

- Oman's Models - In 1991 Oman and Hagemeister introduced a composite measure for quantifying software maintainability. The maintainability Index (MI) is a composite measure that incorporates a number of traditional source code measures into a single number that indicates relative maintainability. As originally proposed by Oman and Hagemeister, the MI is comprised of weighted Halstead measures (effort or volume) [18], McCabe's Cyclomatic Complexity [31], lines of code (NLOC), and the number of comments. Two equations were presented: one that considered the comments and one that did not. The Maintainability Index was originally presented as follows: [35, 36]

  - three-measure $MI = 171 - 3.42 * ln(aveE) - 0.23 * aveV(g') - 16.2 * ln(aveLOC)$ where $aveE$ is the average Halstead Effort per module, $aveV(g')$ is the average extended cyclomatic complexity per module, and $aveLOC$ is the average of lines of code per module.
  - four-measure $MI = 171 - 3.42 * ln(aveE) - 0.23 * aveV(g') - 16.2 * ln(aveLOC) + 0.99 * aveCM$ is the average Halstead Effort per module, $aveV(g')$ is the average extended cyclomatic complexity per module, and $aveLOC$ is the average of lines of code per module, and $aveCM$ is the average number of lines of comments per module. The idea here is that the comments lines will increase the maintainability of the code.

  The original formula was designed for use with procedural languages. Since our case study project is written in Java, we can think of a module as either a package, a class, a method or an overall system comprising a number of these elements. In general, we probably want to look at this measure at a higher level than drill down to find out which parts of the system are contributing most to its low maintainability. It does not really make sense to calculate the maintainability index at the method level as it is far too granular to be of any real use [28]. We reflect this approach by providing calculations of MI at class, package and overall system level.

  Several variants of the maintainability index have evolved over time. One of them is to use the average Halstead Volume instead of the average Halstead effort. Other studies have shown that the maintainability index model was often overly sensitive to the comment measure in the four-measure equations, and thus that portion of the equation was modified to limit the contribution of components in the maintainability index [8, 44]. The modified maintainability index equations look as follows:

  - three-measure $MI = 171 - 5,2 * ln(aveV) - 0,23 * aveV(g') - 16,2 * ln(aveLOC)$ where $aveV$ is the average Halstead Volume per module, $aveV(g')$ is the average extended cyclomatic complexity per module, and $aveLOC$ is the average of lines of code per module.
  - four-measure $MI = 171 - 5,2 * ln(aveV) - 0,23 * aveV(g') - 16,2 * ln(aveLOC) + 50,0 * sin\sqrt{2,46 * perCM}$ is the average Halstead Volume per module, $aveV(g')$ is the average extended cyclomatic complexity per module, $aveLOC$ is the average of lines of code per module, and $perCM$ is the average percentage of lines of comments per module. The idea here is that the comments lines will increase maintainability of the code.

In this research project we use the maintainability index that is provided by the JHawk tool, and there are two versions of the maintainability index - MINC (MI with the No Comment part) and MI (including the comment part). Consideration of the comments in the maintainability index constitutes a big discussion point. To measure the quality of the comments is not a trivial task and, unfortunately, we have not yet managed to find a way of automatically assessing the quality of the comments. For this reason we only take into consideration evaluation of the maintainability index MINC without the comments. The MINC equations used by JHawk looks as follows:

$$MINC = 171 - 3,42 * ln(aveE) - 0,23 * aveV(g') - 16,2 * ln(aveLOC)$$

JHawk uses Java statements rather than lines of code. The authors of the JHawk measurement tool claim that using statements rather than lines of code is the better choice, and we agree with the authors on this point. More information can be found in [26].

- McCall's Model This is one of the earliest models that was presented by Jim McCall and his colleagues [32]; we presented this model in Appendix B. According to this model, maintainability can be measured by combining five criteria:

  - Consistency: Use of uniform design and implementation techniques and notation throughout a project [34].

    $$CONS = 0,7 * AVLCOM + 0,3 * AVUWCS$$

    .

  - Conciseness: Compactness of the source code, in terms of lines of code [34].

    $$CONC = 0,9 * AVNOS + 0,1 * AVUWCS$$

  - Self-descriptiveness: Attributes of software that provide the documentation that explains implementation of the components [34].

    $$SELD = AVCCML$$

  - Simplicity: Ease with which the software can be understood and tested.

    $$SIMP = 0,4 * AVUWCS + 0,3 * AVRFC + 0,3 * AVLCOM$$

  - Modularity: Provision of highly independent modules.

    $$MODU = 0,4 * AVUWCS + 0,3 * AVCBO + 0,3 * AVEXT$$

- ISO/IEC 9126 Model ISO/IEC 9126 Model The International standard ISO/IEC 9126-3 defines maintainability as a set of attributes that bear on the effort required to make a specified modification (which may include correction, improvements or adoptions of software to environmental changes and modifications in the requirements and functional specification). Maintainability may be evaluated by the following sub-characteristic:

  - Analyzability: Internal analyzability measures indicate a set of software attributes that bear on the effort needed for the diagnosis of failures, or for identification of parts to be modified [20].

    $$ANAL = 0,4 * AVNOS + 0,4 * AVRFC + 0,2 * AVHEFF$$

– Changeability: Internal changeability measures indicate a set of software attributes that bear on the effort needed for modification, fault removal or environment change [20].

$$CHAN = 0,3 * AVNOS + 0,3 * AVCBO + 0,3 * AVEXT + 0,1 * AVHEFF$$

Ayalew and Mguni [3] claim that CBO is a good indicator of changeability.

– Stability: Internal stability measures indicate a set of software attributes that bear on the risk of unexpected effect of modifications [20].

$$STAB = 0,3 * AVNOS + 0,3 * AVCBO + 0,1 * AVEXT$$
$$+0,1 * AVINST + 0,1 * AVPACK + 0,1 * AVLOCM$$

– Testability: Internal testability measures indicate a set of software attributes that bear on the effort needed to validate the modified software [20].

$$TEST = 0,4 * AVNOS + 0,3 * AVCBO + 0,3 * AVRFC$$

4.4 Tools Used in the Measurement

The most important benefit of using a tool in the assessment of the quality of code is that it provides an objective analysis. In this research project we will use Jhawk [26] as a tool for collecting measures. JHawk is a static code analysis tool, i.e. it takes the source code of one's project and calculates the measures based on numerous aspects of the code, e.g. volume, complexity, relationships between class and packages and relationships within classes and packages [26]. The JHawk 5.1 Professional Edition is used in this thesis. The author of JHawk provided both the Standalone and Eclipse [13] plug-in versions of the product in one release package, and we will be using the Standalone version. The author has also given us the possibility of creating own measures and to fully integrate them with JHawk. The JHawk Professional license includes the following features:[26]

- standalone code analyzer which analyzes the Java code and prepares the results to be viewed with the application itself or one exported to HTML, CVS or XML formats;
- Eclipse plug-in includes all of the functionality of the standalone code analyzer;
- Data Viewer product which allows us to view variation in the code over time based on the JHawk Interchange files generated from the standalone and command line versions;
- Command Line version of the JHawk which analyzes Java and code and automatically creates output in HTML, CSV and XML formats (both standard and JHawk Interchange format)

4.4.1 Why use Jhawk as a Measurement Tool?

We studied different tools for collecting measures. Finally, we decided to choose the JHawk tool set for this thesis. In this subsection we present the arguments for this choice:

- the professional versions of JHawk provide one with the ability of creating own measures;
- JHawk is the updated product with support. It had been under development since 1996, when it was initially used to analyze Smalltalk source code;

- comprehensive documentation reduces time spent on learning JHawk;
- it is simple to install and configure and is written entirely in Java;
- JHawk produces an interchange format that can be used by the JHawk Data Viewer to compare measures over time (e.g. different builds);
- it has a rich data export functionality, exporting in XML, CVS, HTML formats;
- the price for a professional license is not too high as compared with other commercial collecting tools;
- under the testing period (two weeks) the JHawk was stable and readable. The Eclipse plug-in was omitted due to problems with Eclipse integration;
- a rich set with pre-implemented measures for each of the examination levels.

4.4.2 Other Alternative Measurement Tools

As we mentioned above, we have been testing other tools for collecting measures. In this subsection we shortly describe them and we present the arguments why we decided not to use them.

- One of the first tools to be tested was SQUANER(Software QUality ANalyzER), a framework for monitoring the evolution of the quality of object-oriented systems. SQUANER connects directly to the SVN of a system, extracts the source code, and performs quality evaluations and fault predictions every time a commit is made by a developer. After quality analysis, feedback is provided to developers with instructions on how to improve their code [16]. SQUANER presents most of the features as Jhawk but the main reason why we omitted the tool was instability during the test period and lack of technical support for the tool.
- The second tool to be tested was SONAR [42]. Sonar is an open platform to manage code quality. As such it covers the seven axes of code quality: *a*) comments; *b*) coding rules; *c*) potential bugs; *d*) complexity; *e*) unit tests; *f*) duplications; *g*) architecture and design . Covering new languages, adding rules engines, and computing advanced measures can be done through a powerful extension mechanism. One of the important plug-ins is the SQALE plug-in which enables Software Quality Assessment based on Lifecycle Expectations for the SONAR project [25, 41, 43]. A plug-in which provides easy-to-use and industrially ready Software Quality Assessment based on Lifecycle Expectations for the SONAR is expensive. The too high price of this plug-in was the main reason why we omitted this tool.
- The third and last omitted tool was the Software QUALity Enhancement project (SQUALE) focused on two main aspects: [4, 39]
  - Work on enhanced quality models: *a*) inspired by existing standards (ISO-9126) and approaches (GQM, McCall); *b*) validated and improved by famous researchers who are part of the Squale team; *c*) dealing with both technical and economical aspects of quality .
  - Development of an open-source application that helps assess software quality and improve it over time: *a*) based on third-party technologies (commercial or open-source) that produce raw quality information (like measures, for instance); *b*) using the quality models to aggregate this raw information into high level quality factors; *c*) all of this targeting different languages, including Java, C/C++, .NET, PHP and Cobol.

Due to installation problems and instability during the test period we decided not to use this tool. The last release 7.1 is dated May 26, 2011, which is quite old.

4.5 Chapter Summary

In this chapter we presented four different code module levels of examination, namely: *a*) method; *b*) class; *c*) package; and *d*) system; . Then we presented the software quality measures and models. In Section 4.2 we presented, briefly, some of the software measures that claim to indicate the complexity for the code. In Section 4.3 we presented three different models to compute the maintainability indices. One of the earliest models was presented by Jim McCall - Oman's Models, Maintainability Index and international standard ISO 9126. In the last part of this chapter we presented the tools used in the measurement and some other, omitted alternative tools as well. In the next part of this thesis we present a case study project as well as the details and results from the experimental work.

Part II

Case Study Project

Chapter 5

Experimental Work

5.1 Decryption of the Project

One small enterprise project was developed for the purpose of this experimental work. The project was named E-Invoice. The main idea of the project was to build an enterprise application which could offer an online service for invoicing - one service which could handle multiple companies with many users at once. Fowler [14] described an enterprise class of applications that often have much complex data to work on. These go together with business rules that fail all tests of logic reasoning. There is no precise definition for enterprise applications, but we can give some indication of Fowler's meaning:[14]

- enterprise applications usually involve persistent data. The data need to be persistent because they need to be around between multiple runs of the program;
- there are usually large amounts of data. Moderate systems will have over 1 GB of data organized in tens of millions of records. With this size of data, managing constitutes a major part of the system;
- when we have a system that is Web-based, so that many people access the data concurrently, then ensuring that all of them can access the system properly is a crucial task;
- because we have so much data there will usually be many user interface screens to handle that data;
- enterprise applications usually need to integrate with other enterprise applications that scattered around the enterprise;
- the term "enterprise application" does not always imply a large system. There are many small enterprise systems which have a strategic impact on the organization's products and business operations.

The goal off this experimental work was to build a fully working service as an E-Invoice service. Due to time limitations and many difficulties during the set-up phase we were not able to fulfill the goal. Instead, we implemented four cases based on the main idea of the E-Invoice project but with a minimal set of functionality. All four cases have the same set of functionality and are based on the same technology used in the implementation; only difference between them is that each case includes a different set of design patterns to achieve the goal.

5.1.1 Functional Requirements

In this subsection we present the functional requirements for the case study project. Each of the cases described in the next sections includes all of the functional requirements.

High level functional requirements:

- the system shall handle multiple companies in one service;
- the system shall provide a service for each company that allows access to many users;
- the system shall provide the user with an authentication service;
- the system shall provide the user with an edit profile service;
- the system shall provide an internationalization service;
- the system shall provide the user with a dashboard after successful login;
- the system shall provide the user with an interface with a responsive HTML template;
- the system shall provide the user with the possibility of personalization of the service.

Low level functional requirements:

- the system shall validate all passwords containing upper- and lowercase characters and one number;
- the system shall preserve the current location on the web service during changing of the internationalization language;
- the system shall provide the user with a logging service;
- the system shall provide error logging for administration of the service;
- the system shall provide a valid HTML template.

5.1.2 Technology Used

In this subsection we present the technology which we used for the case study project. The case study project of E-Invoice is a typical three-tier architecture project with a client-server architecture in which the presentation, application processing and data management functions are logically separated. When we use three-tier architecture we can easily make independent changes to each of the three tiers; for example, if we need to upgrade the data management tier then this upgrade will only affect the data management tier.

The three-tier architecture has the following three tiers: [46]

- presentation tier - this tier is responsible for displaying information related to such services as browsing, login or log out. It is a layer which users can access directly, such as a web site;
- application tier - this tier is responsible for controlling an application's functionality by performing detailed processing;
- data tier - this tier is responsible for storing and retrieving information from the database servers. This tier keeps data neutral and independent from the application servers or business logic. In this way we can also improve scalability and performance.

Wikipedia [46] claims that the three tier may seem similar to the *Model View Controller* pattern. They point to a fundamental rule in the three-tier architecture - the client tier never communicates directly with the data tier - as, in the three-tier model all communication must pass through the middle tier. Conceptually, the three-tier architecture is linear. However the *Model View Controller* architecture is triangular: the view updates to the controller, the controller updates the model, and the view gets updates directly from the model [46].

We have used the following technology for each of the tiers:

- presentation tier - presentation is the content rendered by the browser on the client side;
- application tier - Java Enterprise Edition, more details about JavaEE can be found in [37]. As an application server we used Apache Tomcat; more details about Apache Tomcat can be found in [2];
- data tier - a relational database management system (RDBMS) Mysql that runs as a server providing multi-user access to a number of databases. More details about the Mysql data base server can be found in [38].

In this case study project we also used an object-relational mapping (ORM) library for the Java language which provides a framework for mapping an object-oriented domain model to a traditional relational database. Hibernate solves object-relational impedance mismatch problems by replacing direct persistence-related database accesses with high-level object handling functions [45]. More details about the Hibernate project can be found in [9].

5.2 Experimental Methodology

5.2.1 Selection of Design Patterns

In this subsection we offer a selection of the design patterns we use in this research. The study of design patterns offers a wide range of patterns, but for the purpose of this thesis we only study some of them, namely, the design patterns that we found useful in implementation of the case study project. In this thesis we focused the discussion on the architecture of the *Model View Controller* software pattern. In Chapter 2 we presented the design patterns for *Model View Controller* architecture. In Figure 5.1 we present how four design patterns collaborate and which of them is which part of the *Model View Controller* architecture.

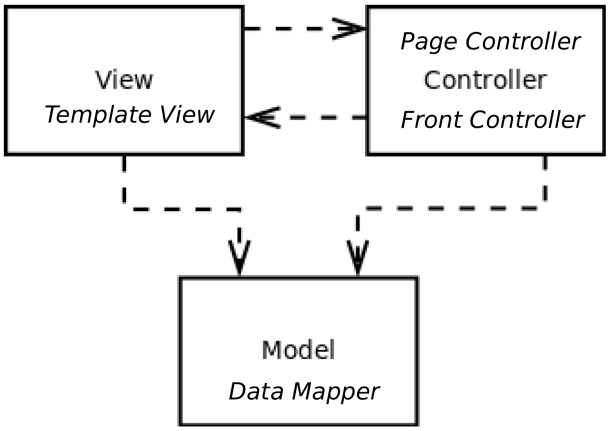

Figure 5.1: Model View Pattern with different design patterns

The choice of this set of patterns was based on several criteria. First, due to time limitation and space limitation of this thesis we had to reduce the scope of the research. Second, during implementation of the the case study project we found that these patterns fit World Wide Web

Enterprise applications best in the case of Java Enterprise. We had worked with these kinds of patterns before, and our interest in assessing the impact of using them was also a criterion.

In Table 5.1 we present four cases in which we try to assess the impact of enterprise application architecture with different sets of design patterns. In each of the four cases we implement and use the *Data Mapper* [14] and we do this for simplicity's sake. We study the impact of using design pattern-based systems by combining different controllers with *Template View* or without it.

Set with design patterns	Design patterns: Data Mapper	Template View	Page Controller	Front Controller
Case 1	✓	✓	✓	
Case 2	✓		✓	
Case 3	✓	✓		✓
Case 4	✓			✓

Table 5.1: Relationship between case and design patterns.

5.2.2 Selection of Quality Measures for Comparison

For the purpose of this research we collected many measures for different levels of examination. As we have mentioned earlier, all of the collected measures are presented in Appendix A and Appendix B. However, when we compare the cases we do not use all of the collected measures. In this subsection we offer a selection of quality measures for comparison. We select a set of measures which, in our opinion, indicate complexity and maintainability. When we compare the cases we analyze the results for the whole system, on the system, the class and method level of examination. The package level of examination is omitted. Why do we not compare all of the available measures? First, the scope of this research will not allow it. Second, not all of the measures are designed to indicate quality for whole system. Some of them are good indicators, but only for a particular level of examination; class or method, e.g. CREF. Some of them are used as a base measure (NAND) for the derived measure (HDIF).
When we compare measures for the class or method level of examination then we need to convert the measure somehow to fit the whole system. The method is the average using arithmetic mean: values of all modules are summed up and divided by the number of modules.

We compare average measures on the method level, which means values of all methods are summed up and divided by the number of methods. The measure converted in this way is: *a*) AVCC.

Some of the measures are designed to indicate quality at the class level of examination. We compare the average measures at the class level, which means values of all classes are summed up and divided by the number of classes. Measures converted in this way are: *a*) AVUWCS; *b*) AVINST; *c*) AVPACK; *d*) VARFC; *e*) AVCBO; *f*) AVLCOM; *g*) AVEXT; and *h*) AVCCML.

Comparing average measures at the system level means we collect cumulative measures at the system level and then divide them by the number of packages, classes and methods.

The measures converted in this way are: *a*) AVNLOC; *b*) AVNOS; *c*) AVHFF; *d*) AVHBUG; *e*) AVHLTH; and *f*) AVVOL.

From our point of view it can be interesting to investigate some measures in the total value, which means the cumulative measure for the whole system for all levels of examination. The measures which are in the total value are: *a*) NLOC; and *b*) NOS.

by using collected measures for complexity, we build three models for maintainability:

- Maintainability Index without the comment part - MINC. We chose a version without comments because we believe that measuring the quality of the comments is not a trivial task and the comments are subjective. MINC uses the Halstead measures, which give a sense of how complex the individual lines of code (or statements) and McCabe's cyclomatic complexity plus some other factors relating to the number of statements are.
- McCall's model which combines five criteria to measure maintainability. This model is one of the earliest models for maintainability and is well established.
- The ISO/IEC 9126-3 quality model was proposed as an international standard for software quality measurement in 1992. It is a derivation of the McCall's model.

We chose three different models because we wanted to see the differences in the results between them. Since they use different base measures to indicate the maintainability, the results can differ. However, base measures have different ranges of values, and to use them in the maintainability formulas it is necessary to have all the measures having the same range (in interval [0; 1]). This could be achieved by normalization. We use the following formula for normalization:

$$Normalized(e_i) = \frac{e_i - E_{min}}{E_{max} - E_{min}}$$

where:

- E_{min} the minimum value for variable E;
- E_{max} the maximum value for variable E

5.3 Tools Used In The Experiments

In this section, several tools that were used for the experiments are presented. We present tools for collecting measures, developing applications and processing collected measures values.

As we presented in Section 5.1, we implemented four cases based on the main idea of the E-Invoice project. As a development tool we used The Spring Tool Suite [40] which is built on top of the Eclipse [13] platform. This tool is customized for developing Spring applications. We do not use Spring framework, although it can be used for a general purpose Java Enterprise application. The tool suites provide a ready-to-use combination of language support, framework support, and runtime support, and combine them with existing Java Enterprise tooling from Eclipse. We chose STS due to the support technology used in this experimental project and its out-of box solution for rapid Java Enterprise development. However, during the development process we found many difficulties in customization of the STS tool.

For measurements we used the JHawk static code analysis tool. We describe the tool and the reasons why we chose this tool in Section 4.4.

Collected measures are exported to CSV [1] files and imported to the LibreOffice Calc [12], which helps us analyze our data and then use it to present our final output. We use LibreOffice Calc because of our familiarity with this tool, which makes it easier to enter complex formulas.

[1] Comma-separated values (CSV), the file stores tabular data (numbers and text) in plain-text form.

5.4 Chapter Summary

In the first part of this chapter in Section 5.1 we presented in details the experimental enterprise project E-Invoice. Based on Fowler's meaning we tried to present some indications of the enterprise application. It is important to remember that the term "enterprise application" does not always imply a large system. The are many small enterprise systems which have a strategic impact on the organization's products and business operation. Due to a time limitation and difficulties during the set-up phase we could not build a fully working E-Invoice service. Instead, the four cases based on the E-Invoice project were developed. We presented the functional requirements and technology used for all four cases. In the second part of this chapter, in Section 5.2, we introduced the experimental methodology. Subsection 5.2.1 offers a selection of the *Web Presentation* design patterns we used in this research; namely, *a*) *Model View Controller* architectural pattern; *b*) *Page Controller* pattern; *c*) *Front Controller* pattern; and *d*) *Template View* pattern.
We explained why we chose this set of patterns, and in Table 5.1 the relationship between the case studies and the design pattern is described. Next, in Subsection 5.2.2 we presented a selection of quality measures for the comparison. We explained why we chose those measures and how we compared them for the whole system. In the last section 5.3 we described the tools for collecting measures (JHawk), developing applications (STS) and processing data (LibreOffice Calc).

In the next chapter, Chapter 6. We present all of the four cases in detail. We analyze the comparison of the results and try to assess the impact of using design patterns of enterprise application architecture in terms of complexity and maintainability.

Chapter 6

Experimental Process

6.1 Introduction

For the purpose of this thesis we developed four small enterprise applications which differ in sets of used design patterns. We present all four cases in the following sections and point out the implementation differences in each of the case. They differ in implementation, because they use different sets of design patterns, but they have these things in the common:

- all four cases are based on the functional requirements presented in Subsection 5.1.1 and the technology described in Subsection 5.1.2;
- they use the *Data Mapper* pattern [14] which is implemented in the pl.arturkb.EInvoice.Utils.Hibernateutil class;
- they use the model part of the *Model View Controller* pattern architecture which is implemented in the pl.arturkb.EInvoice.Beans.Model.User class;
- they use two types of business logic, domain logic and application logic. This concept of separation was introduced by Evans [10]. Evans claimed that domain logic is logic that corresponds to the actual domain. In our case this is an invoicing application, then the domain rules should be rules regarding invoicing, posting, taxation, etc. We put the domain logic in the model layer. However, there are rules such as, for example, different type of export/import - which has nothing to do with the actual domain. This kind of logic is application logic, and we place it in the controllers directly.
- they use an *Intercepting Filter* [1]. We use filters to authenticate users, logging and internationalization. This pattern is implemented in the following packages and classes:
  - pl.arturkb.EInvoice.Beans.Filter.InternatioliziationFilter;
  - pl.arturkb.EInvoice.Beans.Filter.InternatioliziationFilterSecure;
  - pl.arturkb.EInvoice.Beans.Filter.InternatioliziationLoginFilter.

6.2 Case 1

6.2.1 Introduction

In this subsection we introduce the implementation details of case study 1. Our goal in this case study was to combine two *Web Presentation* patterns in the small enterprise application with a minimal set of functionality. With this approach we can achieve *Model View Controller* architecture. When we use *Model View Controller* architecture we can separate the business logic and data access layer from the presentation layer.

6.2.2 Page Controller Pattern

The idea of the *Page Controller* pattern is thoroughly presented in Section 2.4. This pattern is the controller part of the *Model View Controller* architecture. In this case study implementation, we use the *Page Controller* together with the *Template View* pattern. The main responsibility of the *Page Controller* is to:

- decode the URL and extract data from the request;
- create and invoke any model objects to process the data. All of the data from the request should be passed to the model so that the model objects do not need any connection to the request;
- determine which view should be used to display model information on it.

The *Page Controller* pattern is implemented in the following packages and classes:

- pl.arturkb.EInvoice.Controller - this package contains classes that are responsible for internationalization and error logging services:
  - ChangeLanguage - this class is responsible for the internationalization service;
  - Log4jInit - this class is responsible for error logging for the administration;
- pl.arturkb.EInvoice.Controller.Dashboard - this package contains classes with controllers for a dashboard:
  - Index - this class is responsible for the main dashboard service;
- pl.arturkb.EInvoice.Controller.User - this package contains classes with controllers for the user:
  - Edit - this class is responsible for providing the user with an edit profile service;
  - Login, Logout - these classes are responsible for providing the user with an authentication service.

6.2.3 Template View Pattern

The *Template View* pattern is precisely presented in Section 2.6. The *Template View* pattern is the view part of the *Model View Controller* architecture. In this case study implementation we use it together with the *Page Controller* pattern. The *Template View* pattern is implemented in the following packages, classes and files:

- pl.arturkb.EInvoice.UI - this package contains classes which are helper classes for the templates;
- pl.arturkb.EInvoice.Beans.Alert - this package includes classes which are helper classes for the templates;
- WEB-INF - this directory contains JSP files and directories with *Template Views* for different *Page Controllers*. Since the source code in this directory is not written in pure Java, we were unable to analyze the source code with the JHawk tool:
  - Dashboard - this directory contains the *Template View* JSP file for the main dashboard service;
  - User - this directory contains *Template Views* JSP files for user services;
  - Template - this directory includes *Template Views* JSP files for the main window interfaces;

6.3 Case 2

6.3.1 Introduction

Case study 2 is quite similar to case study 1. As we presented in Section 6.2, we used both the *Page Controller* and *Template View* patterns in case study 1. In case study 2 we omitted the *Template View* pattern and the controllers produced output for the web interface without JSP support. Because we omitted the *Template View* pattern we could not separate the business logic from the presentation layer. Our architecture is no longer *Model View Controller* architecture.

6.3.2 Page Controller Pattern

The main responsibility of the *Page Controller* is to:

- decode the URL and extract data from the request;
- create and invoke any model objects to process the data. All of the data from the request should be passed to the model, so that the model objects do not need any connection to the request;
- produce the view and display model information on it.

The *Page Controller* pattern is implemented in the following packages and classes:

- pl.arturkb.EInvoice.Controller - this package contains classes that are responsible for internationalization and error logging services:
  - ChangeLanguage - this class is responsible for the internationalization service;
  - Log4jInit - this class is responsible for error logging for the administration;
- pl.arturkb.EInvoice.Servlet - this package contains classes with controllers:
  - Dashboard_Index - this class is responsible for the main dashboard service;
  - Edit - this class is responsible for providing the user with an edit profile service;
  - Login, Logout - these classes are responsible for providing the user with an authentication.
- pl.arturkb.EInvoice.Utils - this package contains the utilities classes:
  - UI - helper class for servlet controllers with a common task of producing the view.

6.4 Case 3

6.4.1 Introduction

Case 3 is the third experimental case where the main idea is to combine the *Front Controller* and *Template View* patterns. Using those two design patterns will help us to separate the business logic and data access layer from the presentation layer. By combining these two design patterns we achieved the *Model View Controller* architecture.

6.4.2 Front Controller Pattern

The *Front Controller* pattern is thoroughly presented Section 2.5. In our implementation we use a dynamic version of the *Front Controller* pattern. We take some part of the URL and use the dynamic instantiation to create a command class. The "command" parameter tells the Web handler which command to use. We implement this controller part of the *Model View Controller* architecture with five commands; LoginCommand, LogoutCommand, UnknowCommand, EditCommand, and DashboardCommand. In this case study implementation we use the *Front Controller* pattern together with the *Template View* pattern. The *Front Controller* is structured in two parts:

- the Web Handler. This object is responsible for parsing URL and requests and decides what action to initiate and then delegate to a command to carry out the action;
- a command hierarchy, the commands are implemented as classes rather than server pages. They are responsible for:
  - decoding the URL and extracting data from the request;
  - creating and invoking any model objects to process the data. All of the data from the request should be passed to the model, so the model objects do not need any connection to the request;
  - determining which view should be used to display model information on it.

The *Front Controller* pattern is implemented in the following packages and classes:

- pl.arturkb.EInvoice.Controller - this package includes classes that are responsible for error logging services:
  - Log4jInit - this class is responsible for error logging for the administration;
- pl.arturkb.EInvoice.FrontController - this package contains classes for controllers which are not secured , which means they are available without authentication:
  - ChangeLanguageCommand - this class is responsible for the internationalization service;
  - FrontServelet - this class is the Web Handler;
  - FrontCommand - this is an abstract class for command hierarchy;
  - UnknowCommand - this class is used by the Web Handler when the handler cannot find a command;
  - Login - these classes are responsible for providing the user with an authentication service.
- pl.arturkb.EInvoice.FrontControllerSecure - this package includes classes for controllers which are secured, which means they are accessible with authentication only. We present only classes which differ from pl.arturkb.EInvoice.FrontController :
  - Dashboard_Index - this class is responsible for the main dashboard service;
  - Edit - this class is responsible for providing the user with an edit profile service;
  - Logout - these classes are responsible for providing to the user with an authentication service.

6.4.3 Template View Pattern

The way the *Template View* pattern is implemented in case study 3 is the same as in case 1, and it is presented in Subsection 6.2.3; the only difference is that we use this pattern together with the *Front Controller* pattern instead of the *Page Controller*. This makes no changes to the *Template View* source code.

6.5 Case 4

6.5.1 Introduction

Case study 4 is the last one to be implemented in the experimental work. In this case we use only the *Front Controller* patterns, i.e. the controller produces output for the web interface without JSP support; thus separation of the business logic from the presentation layer does not exist. This architecture is no longer the *Model View Controller* architecture.

6.5.2 Front Controller Pattern

In this case study the *Front Controller* is quite similar to the implementation of the *Front Controller* from case study 3, which is presented in Subsection 6.4.2. There is a small but very important difference, namely, the *Front Controller* produces the view and displays model information on it instead of sending it to the *Template view*. To manage this extra amount of work we changed the implementation of the *Front Controller* from Subsection 6.4.2 and added an extra code in the following class :

- pl.arturkb.EInvoice.Utils.UI - helper class for controllers with a common task of producing the view.

6.6 Comparison and Discussion

In this section we compare the four implementations of the same system but with a different set of *Web Presentation* patterns. We assess the impact of using design patterns of the enterprise application architecture. With this comparison we develop a quantitative approach for measurement of the impact of using design patterns on maintainability and complexity in the enterprise application architecture.

For almost all of the measures a lower value meant a better value. There are three measures for which a higher value is a better value: *a*) Average number of comments lines in the module (AVCCML), a higher number of comments lines suggests a better value of AVCCML, but it says nothing about the quality of lines. *b*) Maintainability Index has a better value if the value is higher. The range for this measure is $[0, 171]$. *c*) Self-descriptiveness criterion for McCall's maintainability model also has a better value if the value is higher. Since the formula for this criterion uses AVCCML directly, this criterion says nothing about the quality, only about the quantity. This is the reason why we only use a 4% weight for yhe final McCall's maintainability model.

All base and derived measures for complexity and maintainability are described in detail in Appendix A and Appendix B.

6.6.1 Case 1 vs Case 2

As can be seen from Table 5.1, case 1 implements both the *Page Controller* and *Template View* patterns. In the second case we implemented only the *Page Controller* pattern. With this

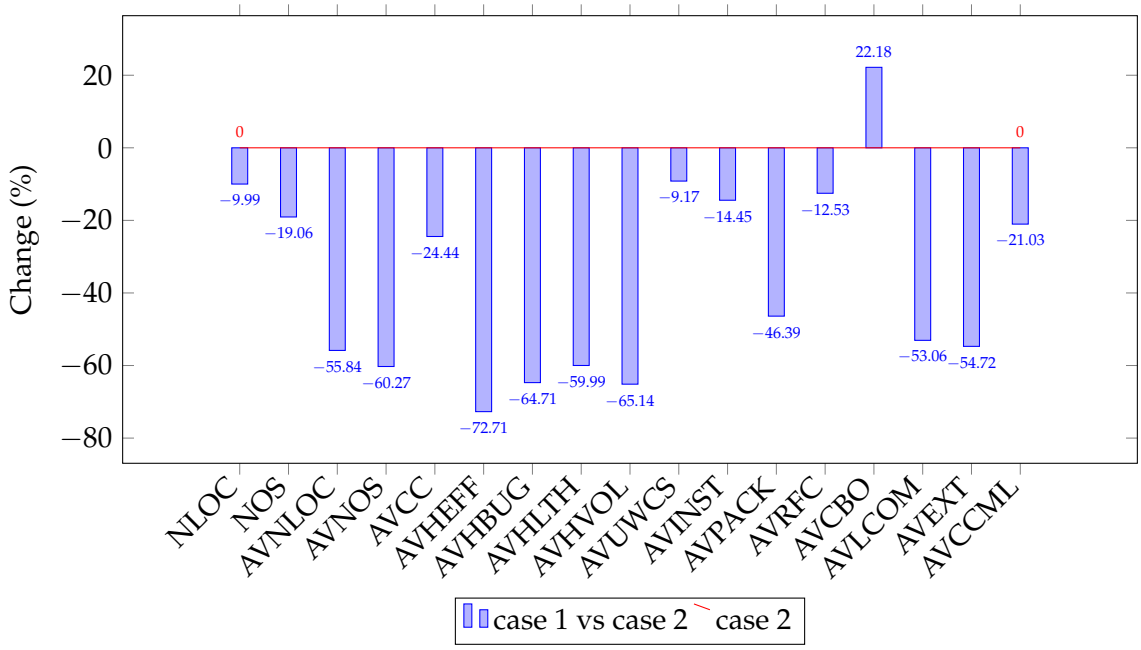

Figure 6.1: Case 1 vs case 2 complexity measures comparison

comparison we try to assess the impact of using the *Template View* pattern of the enterprise application architecture. Using both the *Page Controller* and *Template View* patterns makes the architecture of case study 1 *Model View Controller* architecture. A comparison of the results is presented in Figures 6.1, 6.2 and Figure 6.3.

It can be seen from Figure 6.1 that by using the *Page Controller* and *Template View* patterns together we significantly reduced almost all the complexity measures. This means that we also reduced complexity. The AVCCML value is reduced by 21.03%, but this does not mean that case study 2 has a better value than case study 1. We cannot say anything about the quality of the comments; this only says that case 2 has more comment lines on average than case 1. However, the value of AVCBO increased by 22.18%, and the measure of coupling determined testability and modularity.

From the results presented in Table 6.2 we observed that by using the *Page Controller* and *Template View* patterns together we strengthen Maintainability Index by 21.60%. According to the evaluation model, which is presented in Appendix B.2, case 1 with a value of 115.52 has a poor Maintainability Index. Case 2 with a value of 95 has also a poor Maintainability Index.

Figure 6.2 shows that by using the *Page Controller* and *Template View* patterns together we can reduce McCall's maintainability factor MCC by 32%. Four values of criteria (CONC, CONC, SIMP, MODU) have better values than in case 2. SELD is 20.21% lower than in case study 2, which means that case study 1 has fewer comments on average than case 2, but this says nothing about the quality of the comments.

As can be seen from Figure 6.3, the ISO/IEC 9126 quality maintainability characteristic is lower than in case 2 by 32%. All of the sub-characteristics are much lower than in case 2.

By using the *Page Controller* and *Template View* together we got *Model View Controller* architecture with better maintainability than in case 2, MINC by 21.60%, MCC by 32% and ISO by 32%. The complexity measures also show that case 1 has better complexity than case 2. However, it is important to emphasize that the JSP files used in case 1 were not analyzed, so they had no influence on the collected measures. Only one measure has a negative impact on quality, namely AVCBO, which can reduce modularity and testability. Still, we can safely say that the *Model View Controller* architecture with the *Template View* pattern as a view part of it has a significant positive impact on both complexity and maintainability in the enterprise

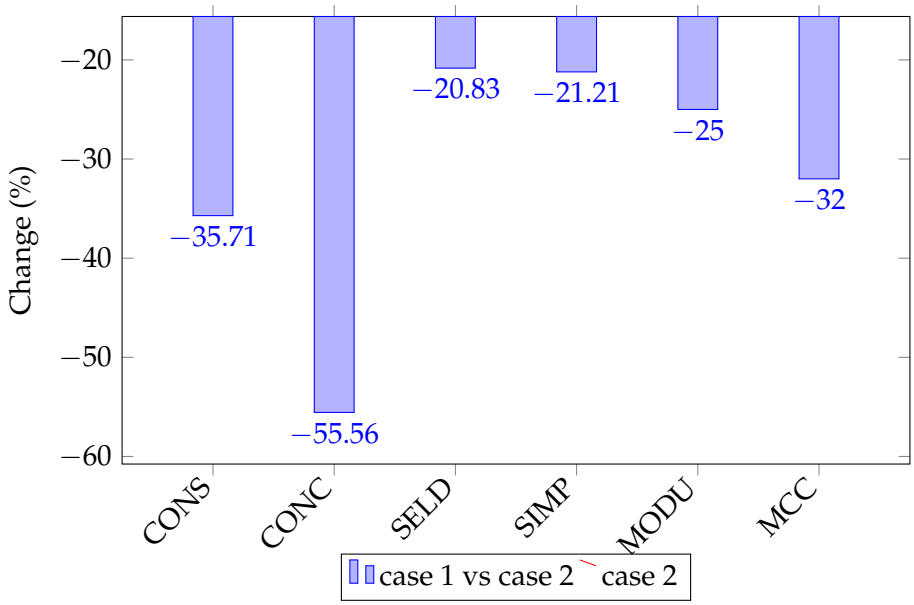

Figure 6.2: Case 1 vs case 2 McCall's maintainability model measures comparison

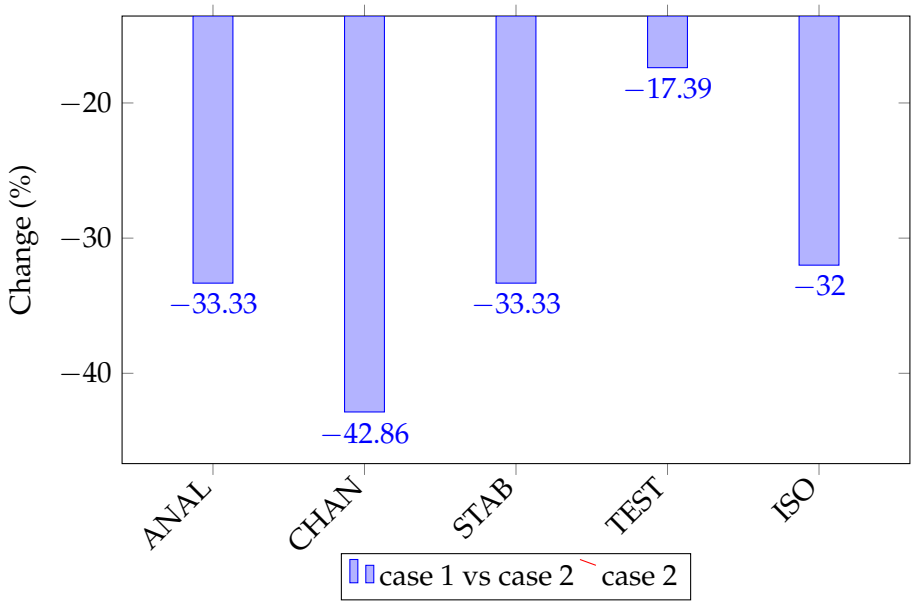

Figure 6.3: Case 1 vs case 2 ISO/IEC 9126 maintainability model measures comparison

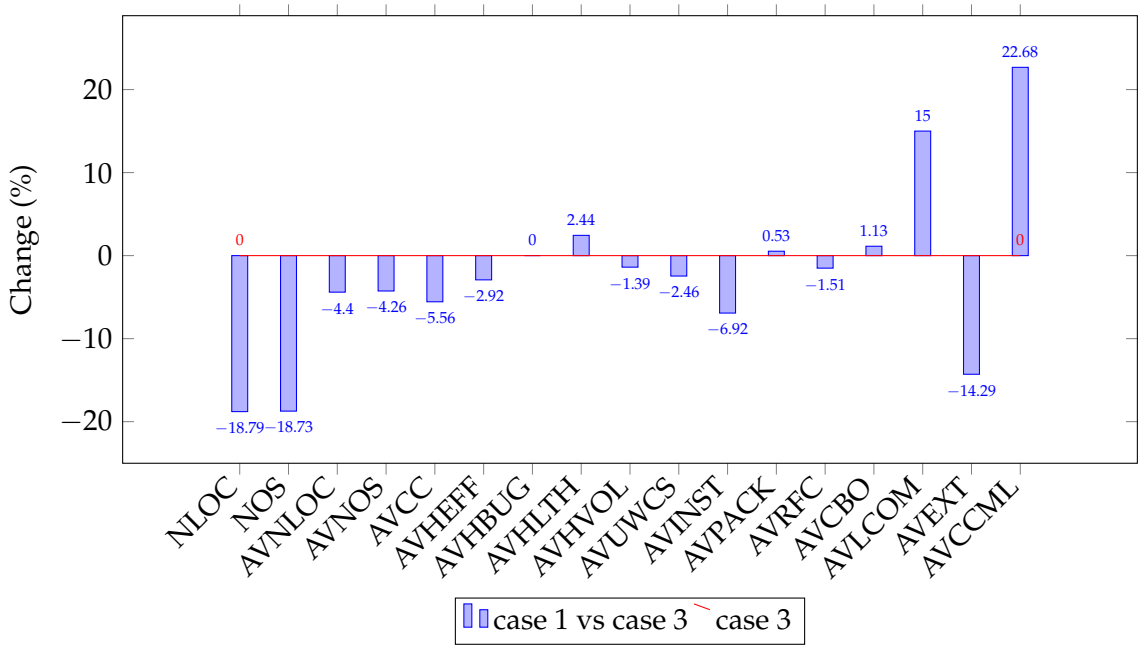

Figure 6.4: Case 1 vs case 3 complexity measures comparison

architecture.

6.6.2 Case 1 vs Case 3

The main idea of this comparison is to see the impact of using two different controller patterns with the same implementation of the *Template View* pattern of enterprise application architecture. We present a comparison of the results in the following Figures: 6.4, 6.5, and 6.6.

It can be seen from Figure 6.4 that by using the *Page Controller* instead of the *Front Controller* pattern we slightly reduce many complexity measures. The NLOC, NOS, AVNLOC, AVNOS, AVCC, AVHEFF, AVHVOL, AVUWCS, AVINST, AVRFC and AVEXT values are lower in different percentages, which means we slightly reduce the complexity. The AVCCML value increased by 22.68%, but this still means that case 1 has a better value than case 3. We cannot say anything about the quality of the comments. AVHBUG stays at the same level as in case 3. The values of AVHLTH, AVPACK and AVCBO increase slightly in case 1, and this makes case 1 slightly more complex than case 3. However, the value of HVLCOM increased by 15%. The lack of cohesion implies that the classes should probably be split into two or more sub-classes [7]. If we look in Appendix C at the class level examination, the highest value of the lack of cohesion LCOM is 1, and classes with a value over 0.5 are generally viewed as bad. They could probably be subdivided into two or more subclasses with increased cohesion.

Next, we compare three different maintainability models. First we look at the MINC. Table 6.2 shows that case 1 has a marginally higher value of MINC than case 3. The difference is 0.03% with value 115.52 for case 1. According to our evaluation model for MINC, both cases have poor Maintainability Indexes. The second maintainability model which we want to analyze is McCall's maintainability model MCC. The only criterion which is reduced (6.9%) is MODU. The SELD criterion value is 22.58% higher for case 1, and this also means that this criterion has a better value in case 1 than in case 3. For case 1 the only criterion which has a worse value than in case 3 is CONS. The CONS criterion value is 5.88% higher for case 1 than for case 3. Overall, Figure 6.5 shows that by using *Page Controller* instead of *Front Controller* we obtained a system with a 5.56% better McCall's factor. The third and last maintainability model is the ISO/IEC 9126 model. For case 1, two sub-characteristics have a lower value than for case 3, which means that case 1 has better values than case 3. However, the ISO quality

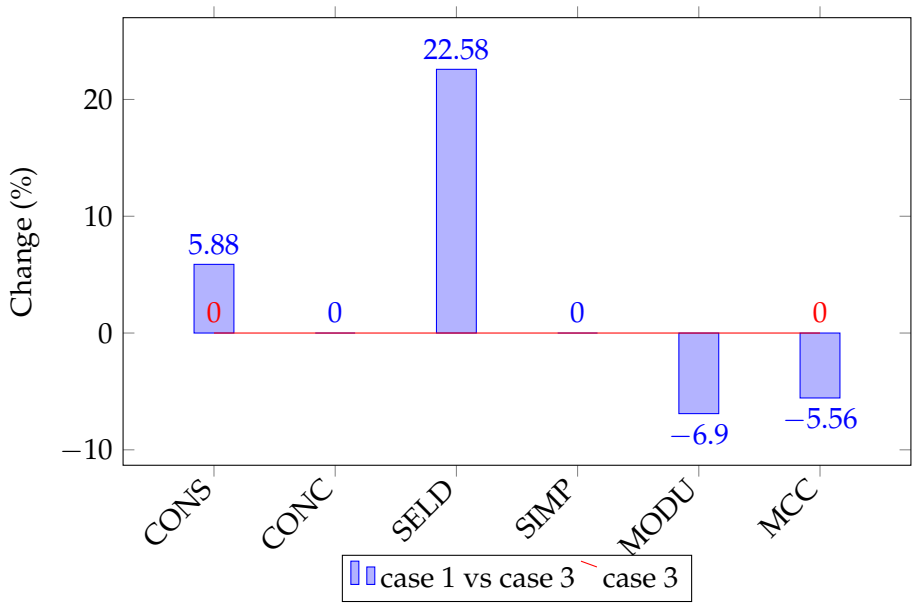

Figure 6.5: Case 1 vs case 3 McCall's maintainability model measures comparison

maintainability characteristic stays at the same level as it is in case 3.

By summarizing the comparison of results, we can say that by using the *Page Controller* pattern instead of the *Front Controller*, we obtained a system where the complexity possibly stays at the same level. Measures such as AVCC, AVUWCS, and AVRFC are marginally lower. However, measures such as AVLCOM and AVCBO are higher. In our opinion it is not clear whether case 1 has a lower or higher complexity. The differences between them are too small to make a clear statement. When we analyze the comparison of results for all three maintainability models we can observe that MINC is marginally better for case 1. MCC is slightly better, but only because we are taking into consideration comments as a base measure for SELD. The ISO maintainability model shows that the differences between those two models are too small to make an impact on the results. By taking all of the three maintainability models into consideration we can say that by using the *Page Controller* pattern instead of the *Front Controller* we obtained a system where maintainability possibly stays at the same level. As we presented in Section 2.5, it is proposed to use the *Front Controller* in situations where the amount of controllers is quite large, as the *Front Controller* is a more complicated design than the *Page Controller*. In our situation the experimental project is a relatively small project, and the trade-off from using the *Front Controller* is not achieved. Future research can deal with a study where the amount of controllers is significantly larger.

6.6.3 Case 2 vs Case 4

In Subsection 6.6.2 we presented and discussed a comparison of the results for the *Model View Controller* architecture. Our conclusion from the comparison is that by using the *Page Controller* pattern instead of the *Front Controller* we obtained a system where complexity and maintainability possibly stayed at the same level. In this subsection we present and discuss a comparison of the results for the *Page Controller* versus the *Front Controller* pattern, but since we do not use the *Template View* pattern in both cases, the design architecture in both cases is not *Model View Controller* architecture. With this comparison of the result we try to assess the impact of using different controller patterns in the non *Model View Controller* architecture. We present a comparison of the results in the following Figures: 6.7, 6.8, 6.9.

First we try to analyze Figure 6.7. Case 2 has all of the Halstead measures, at least 20.98% worse values than in case 4. The Halstead measures give a sense of how complex the individual

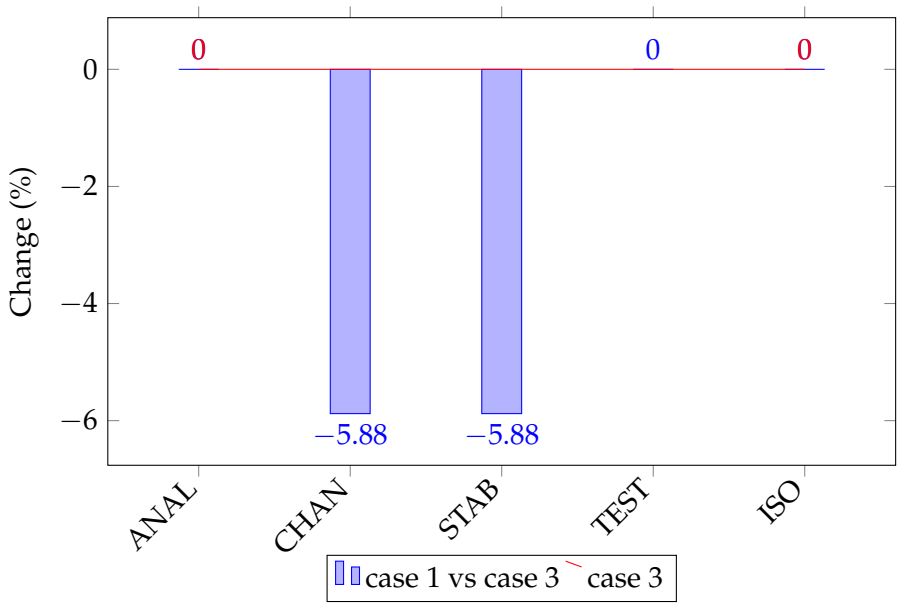

Figure 6.6: Case 1 vs case 3 ISO/IEC 9126 maintainability model measures comparison

lines of code (or statements) are. By using the *Page Controller* pattern instead of front *Front Controller* we also increase McCabes Cyclomatic Complexity AVCC by 1.69%. The Object-oriented measure also increases when we use the *Page Controller*, namely: *a*) AVUWCS by 1.8%; *b*) AVRFC by 2.05%; *c*) AVLCOM by 32.43%; and *d*) AVCBO by 1.03%.

It can be seen from Table 6.1 that MINC is 3.28% lower for case 2 than for case 4. According to the evaluation model, both cases have a poor Maintainability Index with MINC values below 116.

Figure 6.11 shows that by using the *Page Controller* pattern instead of the *Front Controller* pattern we increase McCall's maintainability factor index by 8.7%. Most all of McCall's maintainability criteria values are better in case 4 than in case 2. Only SELD is better in case 2 than in case 4. However, we do not pay much attention to the number of comments lines. When we analyze Figure 6.9, we observe that all four sub-characteristics of the ISO/IEC 9126 maintainability model have higher values in case 2 than in case 4. The ISO quality maintainability characteristic for case 2 is 8.7% worse than for case 4.

This comparison clearly presents that by using the *Front Controller* pattern instead of the *Page Controller* pattern in non-MVC architecture we obtained system which is less complex, but the size of the system is larger. Both measures, NLOC and NOS, are higher for case 4 than for case 2. All three maintainability models also confirm that case 4 has better maintainability. Taking into consideration the complexity and maintainability measures, we can safely make the statement that the overhead from the *Front Controller* pattern pays off when we compare case 2 and case 4.

6.6.4 Case 3 vs Case 4

In Subsection 6.6.1 we presented and discussed the comparison of results for two cases:

- Case 1 where we implemented both the *Page Controller* and *Template View* patterns, which makes case 1 *Model View Controller* architecture.
- Case 2 where we implemented only the *Page Controller*.

We concluded that by using both patterns together we obtained a system which is less complex and a system which is easier to maintain. In this subsection we try to assess the

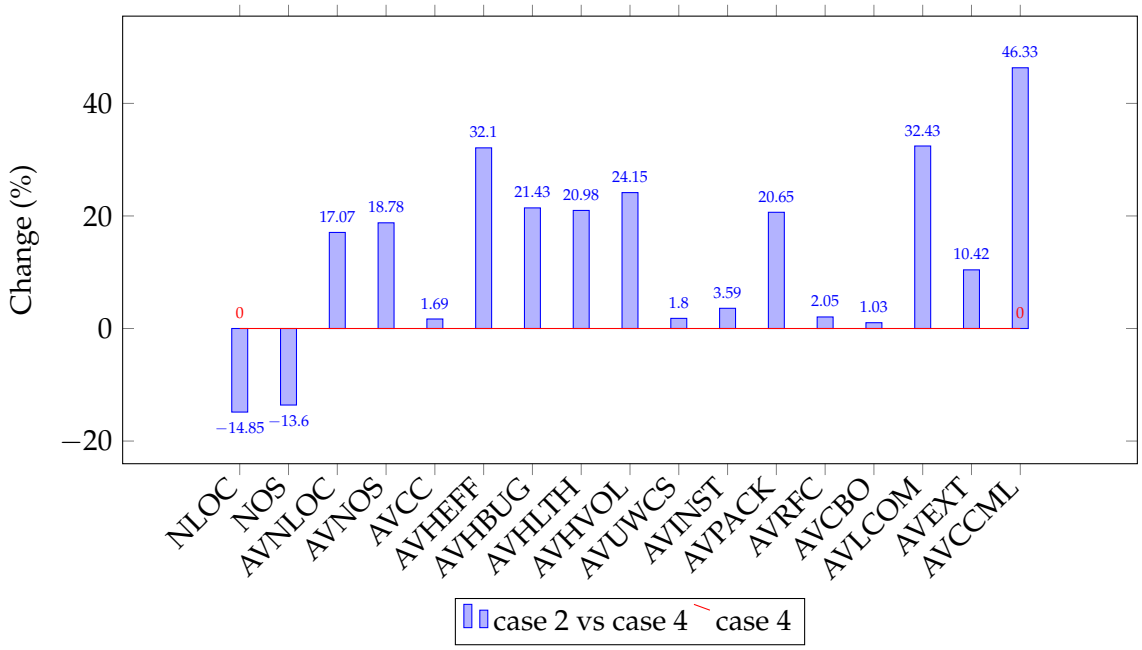

Figure 6.7: Case 2 vs case 4 complexity measures comparison

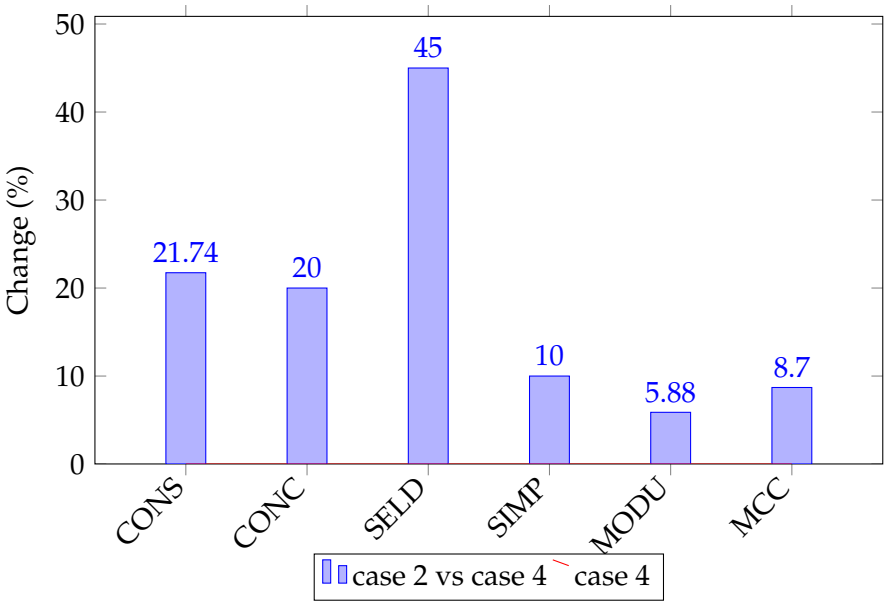

Figure 6.8: Case 2 vs case 4 McCall's maintainability model measures comparison

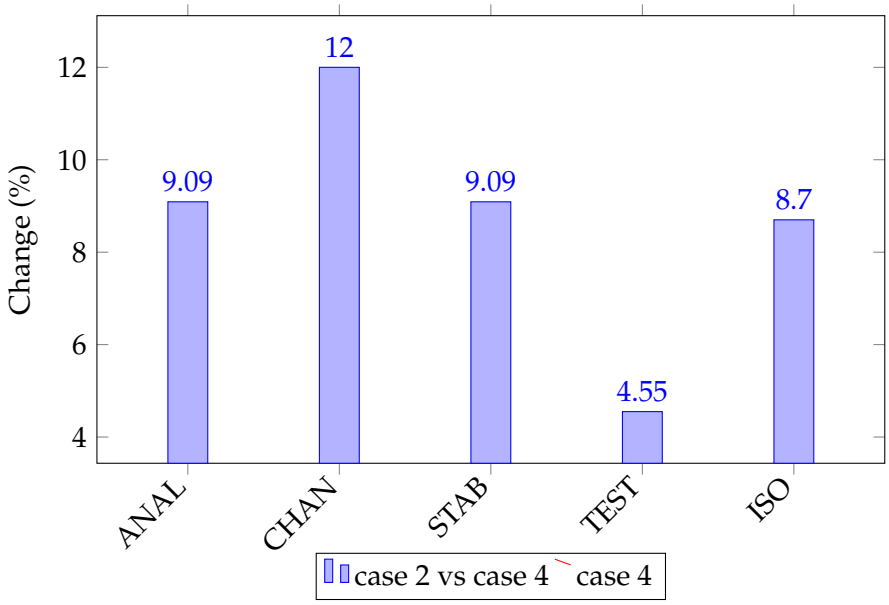

Figure 6.9: Case 2 vs case 4 ISO/IEC 9126 maintainability model measures comparison

impact of using the *Template View* patterns of enterprise application architecture, but this time we use the *Front Controller* pattern instead of the *Page Controller*.

By analyzing Figure 6.10 we clearly see that by using both patterns together

- We significantly reduced all Halstead measures. We also reduced McCabes Cyclomatic Complexity AVCC by 18.64%, which is quite a lot.

- The size of the code on average and the total are also reduced *a*) NLOC by 5.62%; *b*) NOS by 13.95%; *c*) AVNLOC by 45.92%; *d*) AVNOS by 50.17%; *e*) AVUWCS by 5.21%.

- Most of the object-oriented measures are reduced: *a*) AVRFC by 9.36%; *b*) AVLCOM by 45.95%;
  Only AVCBO increased by 22.07%

Table 6.2 shows that by using the *Front Controller* pattern and the *Template View* pattern together we strengthen the Maintainability Index MINC by 17.58%. According to the evaluation model, case 3 and case 4 have poor MINC.

Moving on to the next two maintainability models, we can observe that both of them have better values in case 3 than in case 4. Figure 6.11 shows that most of the criteria for this model reduce in case 3 relative to case 4. The only criterion which increases in case 4 relative to case 3 is SELD, but this only says that case 4 has more comments lines. As we presented in the previous comparisons, we do not pay much attention to this measure due to difficulties in measuring the quality of the comments lines. Overall, by using both design patterns and making case 3 *Model Vie Control* architecture we reduce MCC by 21.74%. In Figure 6.12 we observe a similar reduction in case 3 relative to case 4. All of the four sub-characteristics have better values in case 3 than in case 4. Overall, by using both design patterns we reduce ISO by 26.09%.

To sum up all the observations for complexity and maintainability, we can safely make the statement that by using the *Front Controller* and *Template View* patterns together and thereby making case 3 *Model View Controller* architecture we obtained a system with better complexity and maintainability.

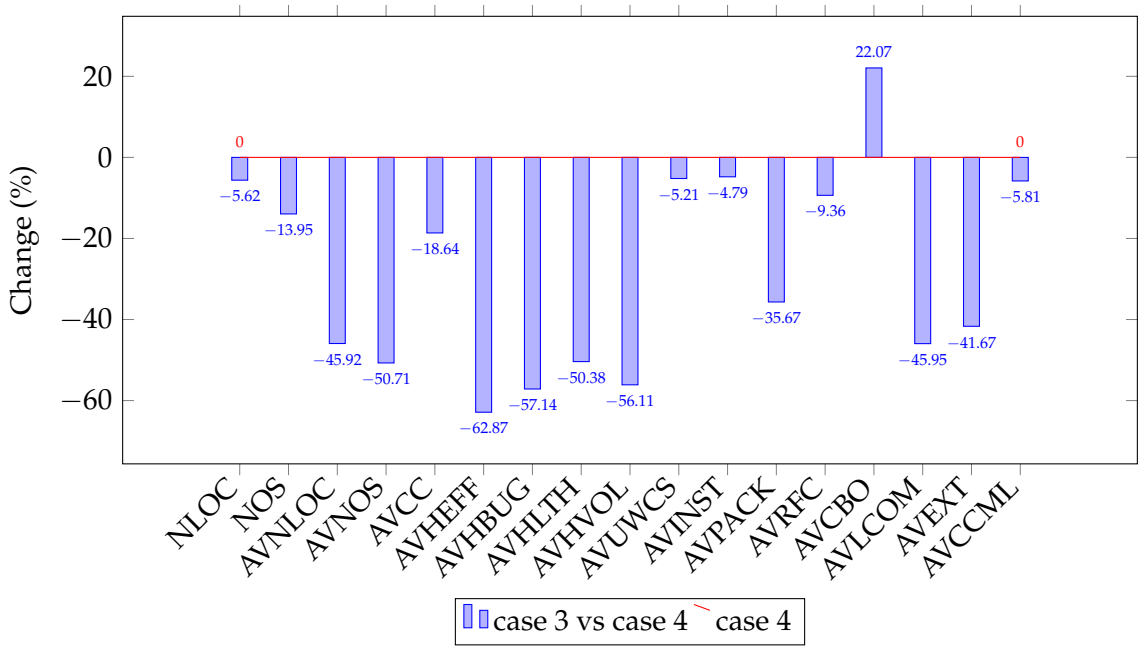

Figure 6.10: Case 3 vs case 4 complexity measures comparison

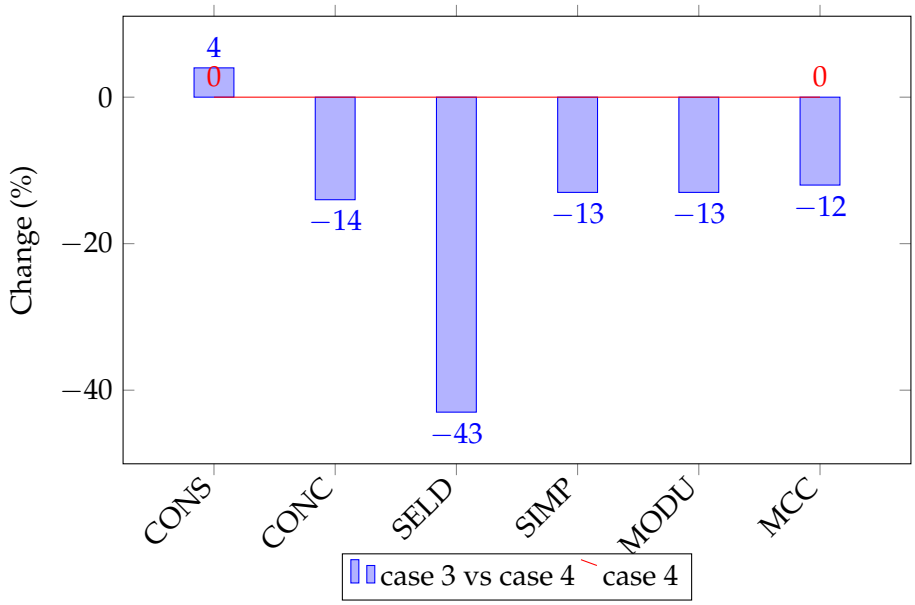

Figure 6.11: Case 3 vs case 4 McCall's maintainability model measures comparison

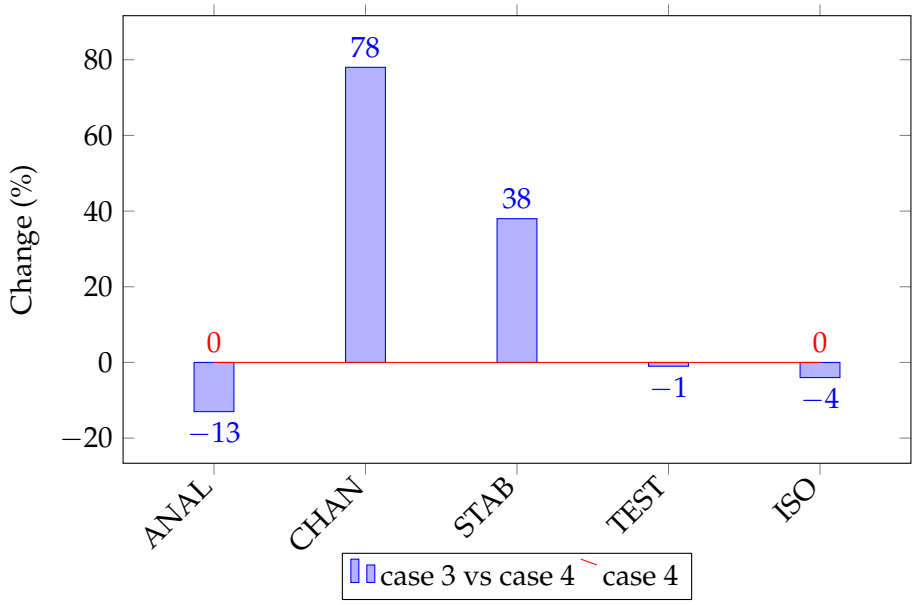

Figure 6.12: Case 3 vs case 4 ISO/IEC 9126 maintainability model measures comparison

6.7 Results

In Table 6.1 we present the values for the complexity measures for the whole system that we compare in the cases. The complete report for all values for the complexity measures is presented in Appendix C. In Table 6.2 we offer values for the maintainability measures for the whole system that we compare. The complete report for the maintainability measures is presented in Appendix D.

6.8 Chapter Summary

In this chapter we presented in detail the experimental process for this case study project. First we introduced a common implementation for all four cases. Next we presented the implementation details for each of the cases where:

- case 1 implements both the *Page Controller* and *Template View* patterns. Case 1 is a *Model View Controller* architecture;
- case 2 implements only the *Page Controller* pattern without the view part, thereby case 2 is not a *Model View Controller* architecture;
- case 3 implements both the *Front Controller* and *Template View* patterns. Case 3 is a *Model View Controller* architecture.
- case 4 implements only the *Front Controller* patterns without the view part, thereby case 4 is not a *Model View Controller* architecture.

After we presented the implementation details for each of the cases we moved on to the comparison part of this chapter. But first we presented for which of the measures the higher values were better values, namely :*a*) AVCCML; *b*) MINC; and *c*) SELD; For the other measures the lower values are better values. Case 1 vs case 2 was the first comparison. We concluded that by using the *Page Controller* and *Template View* patterns together, which is a *Model View Controller* architecture, we obtained the system with better complexity and maintainability. Thus, the impact of using the *Template View* pattern of enterprise architecture is positive.

Table 6.1: Complexity measurement results for the whole system.

Measure	Case 1	Case 2	Case 3	Case 4
NLOC	1063	1181	1309	13870
NOS	807	997	993	1154
AVNLOC	6.52	14.76	6.82	12.61
AVNOS	4.95	12.46	5.17	10.49
AVCC	1.36	1.8	1.44	1.77
AVHEFF	1715	6285.28	1766.56	4757.86
AVHBUG	0.06	0.17	0.06	0.14
AVHLTH	36.77	91.9	37.69	75.96
AVHVOL	181.13	519.58	183.68	418.50
AVUWCS	5.15	6.67	5.28	5.57
AVINST	1.48	1.73	1.59	1.67
AVPACK	3.79	7.07	3.77	5.86
AVRFC	3.91	4.47	3.97	4.38
AVCBO	3.58	2.93	3.54	2.90
AVLCOM	0.23	0.49	0.2	0.37
AVEXT	0.24	0.53	0.28	0.48
AVCCML	25.64	32.47	20.90	22.19

Table 6.2: Maintainability results for whole system.

Measure	Case 1	Case 2	Case 3	Case 4
MINC	115.52	95	115.49	98.22
MCC	0.17	0.25	0.18	0.23
ISO	0.17	0.25	0.17	0.23

Secondly, we compared case 1 vs case 3. Our conclusion was as follows: by using the *Page Controller* pattern instead of the *Front Controller* we obtained a system where the complexity and maintainability possibly stayed at the same level. Next, we compared case 2 vs case 4, which is quite a similar comparison to the previous one. However, in this comparison we used two cases which were not *Model View Controller* architecture in contrast to case 1 and case 3. We clearly concluded that by using the *Front Controller* pattern instead of the *Page Controller* pattern we obtained a system which is less complex and easier to maintain. We also observed that the trade-off from using the *Front Controller* pattern instead off the *Page Controller* was clearly achieved, but only in the non-*Model View Controller* architecture. The last comparison presented in this chapter was a comparison where we compared case 3 vs case 4. In other words, we tried to assess the impact of using the *Template View* of the enterprise application architecture. We presented the following statement by using the *Front Controller* and the *Template View* patterns together, and by making case 3 *Model View Controller* architecture obtained a system with better complexity and maintainability. It is worth to noting that in both comparisons, namely *a*) case 1 vs case 2; and *b*) case 3 vs case 4, by using both design patterns and thereby making the system *Model View Controller* architecture, we obtained a system which is less complex and easier to maintain.

Part III

Discussion and Conclusions

Chapter 7

Future Directions and Conclusions

7.1 Critical Review of the Thesis

We initially intended to use fairly complex and representative enterprise systems with the name E-Invoice in our experimental studies. However, in spite of extensive efforts we were unable to fulfill the goal. Instead, we implemented four cases based on the main idea of the E-Invoice project but with a minimal set of functionality. All four cases have the same set of functionality and are based on the same technology used in the implementation; the only difference between them is that each case includes a different set of design patterns to achieve the goal. This simpler approach implies some limitation on the results. We base each of the results on exactly one example. Most of the examples are simple case examples situated in the context of a larger and more complex system. Another aspect is that all four cases have been written by only one developer with minor experience in developing the enterprise application. This lack of experience can, in our opinion, affect the results. In this thesis the results are measured for a specific system with a specific source code. There is no generalization, as a generalization of the effects of design patterns of enterprise application architecture quality can be investigated as another in-depth study.

We also collected a number of known measures such as *Line of Code*, *Number of Statement*, *Halstead Complexity*, *MCcabe's Cyclomatic Complexity* and so on. We also computed the software maintainability by using measures that were defined previously. We used three different maintainability indexes, such as Oman's Model, McCall's Model and the ISO/IEC 9126-3 Model. It should be mentioned that collecting different quality factors such as *a*) *reliability*; *b*) *testability*; *c*) *portability*; and *d*) *efficiency*, could provide us with results about the quality of the enterprise system. However, we did not use such quality factors due to a lack of time.

7.2 Future Work

The results from the experiments for the four case studies are indicators that *Design Patterns* can influence the quality of design pattern-based enterprise systems. However, we believe that there is still room for future improvements in this research. In the following paragraphs, we outline some of the potential research directions that we believe might be interesting to investigate.

7.2.1 Using More and Different Enterprise Systems

In this thesis we attempted to evaluate the impact of using design patterns of enterprise application architecture. However, the research was based on one simple case study application, an application that is not completed and with limited functionality. More enterprise applications of various sizes could be considered in a continuation of the present

work. One of the possible works could collect a few industry-ready enterprise applications with a similar set of design patterns and assess the impact of using those patterns of the enterprise systems. Another aspect that could be potentially interesting would be a study of the impact of using design patterns of enterprise application architecture in different implementation technology. Java Enterprise was the choice for this study and a comparison with other programming technology may result in interesting results. We suggest technology such as .NET or a similar one.

7.2.2 Using More or Different Design Patterns

For the scope of this research we attempted to assess the impact of using design patterns of enterprise application architecture based on the set of four *Web Presentation* patterns. The size of the set of *Design Patterns* is based on the time limitation and the scope of this thesis. However, each enterprise system has several layers. Layering is one of the most common techniques that software designers use to break apart complicated software systems. Fowlers [14] described the architecture of three primary layers: [14]

- presentation layer: provision of services, display of information(e.g., in Windows or HTML, handling user requests(mouse clicks, keyboard hits), HTTP request, command-line invocations, batch API);
- domain layer: logic which is the real point of the system;
- data source layer: communication with the database, messaging system, transaction managers, other packages.

Web Presentation patterns belongs to the presentation layer. Fowler [14] presents many other *Web Presentation* patterns that were omitted in this research, namely: *a*) *Transform View*; *b*) *Two step View*; and *c*) *Application Controller*.

Other layers also have patterns which could be interesting to examine for the impact of using them on the enterprise systems. One possible future work could be to investigate the impact of using different *Data Source Architectural Patterns*; Fowler [14] introduces four of them: *a*) *Table Data Gateway*; *b*) *Row Data Gateway*; *c*) *Active Record*; and *d*) *Data Mapper*.

All of the four design patterns as presented in the list above belong to the data source layer. In the case study project the *Data Mapper* pattern was used but without any comparison with other *Data Source Architectural* patterns.

7.2.3 Using More Quality Factors

One of the fundamental goals of software engineering is to develop a methodology for the assessment of overall system quality at low cost. We believe that the aspects of complexity and maintainability can play a significant role in this assessment. In this paper, we only used those two factors, due to the time limitation and the scope of this research. Other, software quality factors such as: *a*) *reliability*; *b*) *testability*; *c*) *portability*; and *d*) *efficiency*, can be considered in future research works in order to enlarge the scope of the analysis.

7.3 Conclusion

In this thesis we studied the impact of using design pattern-based systems on the complexity and maintainability of enterprise applications. We used several measures in order to measure these qualities at the implementation level. We presented a collection of measures in Chapter 4 as the well as tools used to perform the measurements. Most of the results show that these

measures are influenced by the presence of different kinds of *Design Patterns*. Four small case studies were used as a tool for this research, and each of the case studies included a diverse set of *Web Presentation* patterns. Overall, we concluded that using *Model View Controller* architecture can in most cases help reduce the complexity. Reduced complexity of the software can lead to software which will at the same time require less maintenance effort. We also learned that design patterns can provide a toolbox of solutions to common problems.

Appendix A

Complexity Analysis Measures

Table A.1: Code Module

System level:	S
Package level:	P
Class level:	C
Method level:	M
Remark:	For levels higher than the method level, measures are cumulative if other functions are not specified.

Table A.2: Number Lines of Code (NLOC)

Entity:	Code Module(M,C,P,S)
Attribute:	Size
Unit:	Number of lines of Code per module (NLOC)
Scale Type:	Ratio
Range:	$[0, \infty)$
Format:	Integer
Remark:	A line of code is any non-blank line in a code file that is not a comment. A line of code with an end of the line comment will be counted as a line of code and a comment [21]. Reduce the value of the numbers lines of code NLOC means that we reduced the amount of work that it would take to understand a particular module.

Table A.3: Cyclomatic Complexity (COMP)

Entity:	Code Module(M)
Attribute:	Complexity
Unit:	Number of execution paths through a method (COMP)
Scale Type:	Ratio
Range:	$[1, \infty)$
Format:	Floating point
Remark:	This is calculated from the number of logical branch points in the method. The method itself is counted as 1 logical branch point. The if, switch, for, while and catch operators count as logical branch points. Within a switch statement each occurrence of the break keyword is treated as a logical branch point. Cyclomatic Complexities over 10 are generally viewed as being bad [21]. Better or reduce the value of Cyclomatic Complexity means that we reduced the amount of work that it would take to maintain a particular module.

Table A.4: Number of methods in the code module (NOMT)

Entity:	Code Module(C,P,S)
Attribute:	Size
Unit:	Number of methods in the code module (NOMT)
Scale Type:	Ratio
Range:	$[1, \infty)$
Format:	Integer

Table A.5: Total Cyclomatic Complexity of all methods in the code module (TCC)

Entity:	Code Module(C,P,S)
Attribute:	Complexity
Unit:	Number of execution paths through all methods in the code module (TCC)
Scale Type:	Ratio
Range:	$[1, \infty)$
Format:	Floating point

Table A.6: Average Cyclomatic Complexity (AVCC)

Entity:	Code Module(C,P,S)
Attribute:	Complexity
Unit:	$AVCC = TCC/NOMT$
Scale Type:	Ratio
Range:	$[1, \infty)$
Format:	Floating point

Table A.7: Number of Comments Lines (NOCL)

Entity:	Code Module(M)
Attribute:	Size
Unit:	Number of comments lines (NOCL)
Scale Type:	Ratio
Range:	$[0, \infty)$
Format:	Integer
Remark:	This is the number of lines of comments in the code including the start and end tokens for the comments if these are on separate lines. An end of line comment on a line that also includes code will be counted as a comment line (it will also be counted as a line of code) [21].

Table A.8: Number of Java statements in the code module (NOS)

Entity:	Code Module(M,C,P,S)
Attribute:	Size
Unit:	Number of Java statements in the code module (NOS)
Scale Type:	Ratio
Range:	$[0, \infty)$
Format:	Integer
Remarks:	A Java statement is defined as a series of Java tokens terminated by a semi-colon.

Table A.9: The Halstead Length of the code module (HLTH)

Entity:	Code Module(M,P,S)
Attribute:	Length
Unit:	$HLTH = NAND + NOPR$
Scale Type:	Ratio
Range:	$[0, \infty)$
Format:	Floating point
Remark:	The Halstead Length of the module. This is the sum of the number of operators plus the number of operands. It is an indicator of module size [21]. A lower value of HLTH means a module which is less complex.

Table A.10: The Halstead Vocabulary of the code module (HVOC)

Entity:	Code Module(M)
Attribute:	Complexity
Unit:	$HVOC = UNAND + UNOP$
Scale Type:	Ratio
Range:	$[0, \infty)$
Format:	Integer
Remark:	The Halstead Vocabulary of the method. This is the sum of the number of unique operators plus the number of unique operands. It is an indicator of method complexity [21]. A lower value of HVOC means a method which is less complex.

Table A.11: The Halstead Volume of a code module (HVOL)

Entity:	Code Module(P)
Attribute:	Size
Unit:	$HVOL = HLTH * \log_2(HVOC)$
Scale Type:	Ratio
Range:	$[0, \infty)$
Format:	Floating point
Remark:	The Halstead Volume of a module is an indicator of module size [21]. A small number of statements with a high Halstead Volume would suggest that the individual statements are quite complex [29].

Table A.12: The Halstead Effort for the code module (HEFF)

Entity:	Code Module(M,C,P,S)
Attribute:	Effort
Unit:	$HVFF = HVOL * HDIF$
Scale Type:	Ratio
Range:	$[0, \infty)$
Format:	Floating point
Remark:	The Halstead Effort for the module is an indicator of the amount of time that it will take a programmer to implement the module [21].

Table A.13: The Halstead Difficulty of the code module (HDIF)

Entity:	Code Module(M)
Attribute:	Complexity
Unit:	$HDIF = UAND * UNOR$
Scale Type:	Ratio
Range:	$[0, \infty)$
Format:	Floating point
Remark	The Halstead Difficulty of a method is an indicator of method complexity [29].

Table A.14: Estimated Halstead Bugs in the code module (HBUG)

Entity:	Code Module(M,C,P,S)
Attribute:	Quality
Unit:	$HBUF = HVOL/3000$
Scale Type:	Ratio
Range:	$[0, \infty)$
Format:	Floating point
Remark	Estimated Halstead Bugs in the method. A Lower value of HBUG means less errors.

Table A.15: Number of operands in the code module (NAND)

Entity:	Code Module(M)
Attribute:	Complexity
Unit:	Number of operand in the code module (NAND)
Scale Type:	Ratio
Range:	$[0, \infty)$
Format:	Integer
Remark:	An operand is a Java token that can have operations carried out on it by other Java tokens (called operators, see Table A.17). Operands include variables, numeric and string literals, special variables such as true, false, null, void, super and this, classes and primitive types and methods [21].

Table A.16: Number of unique operands in the code module (UAND)

Entity:	Code Module(M)
Attribute:	Complexity
Unit:	Number of unique operand in the code module (UAND)
Scale Type:	Ratio
Range:	$[0, \infty)$
Format:	Integer
Remark:	See Table A.15

Table A.17: Number of operators in the code module (NOPR)

Entity:	Code Module(M)
Attribute:	Complexity
Unit:	Number of operators in the code module (NOPR)
Scale Type:	Ratio
Range:	$[0, \infty)$
Format:	Integer
Remark:	An operator is a Java token that is used to carry out an operation on another Java token (called an operand, see Table A.15). Examples of operators would be arithmetic operators and logical operators. Java keywords such as for and while are also operators [21].

Table A.18: Number of unique operators in the code module (UOP)

Entity:	Code Module(M)
Attribute:	Complexity
Unit:	Number of unique operators in the code module (UOP)
Scale Type:	Ratio
Range:	$[0, \infty)$
Format:	Integer
Remark:	See Table A.18.

Table A.19: Number of different classes referenced in the method (CREF)

Entity:	Code Module(M)
Attribute:	Complexity
Unit:	Number of different classes referenced in the method (CREF)
Scale Type:	Ratio
Range:	$[0, \infty)$
Format:	Integer
Remark:	This will include classes that are referenced in variable declarations, argument types, casts, exceptions thrown and caught, instantiations of variables through the new operator and direct references to class methods and variables. The CREF value includes both class and interface references [21].

Table A.20: Number of calls to methods that are not defined in the class of the method (XMET)

Entity:	Code Module(M)
Attribute:	Complexity
Unit:	Number of calls to methods that are not defined in the class of the method (XMET)
Scale Type:	Ratio
Range:	$[0, \infty)$
Format:	Integer

Table A.21: Number of calls to local methods, i.e. methods that are defined in the class of the method. (LMET)

Entity:	Code Module(M)
Attribute:	Complexity
Unit:	Number of calls to local methods (LMET)
Scale Type:	Ratio
Range:	$[0, \infty)$
Format:	Integer

Table A.22: Unweighted class size. (UWCS)

Entity:	Code Module(C)
Attribute:	Size
Unit:	$LMET = NOMT + INST$
Scale Type:	Ratio
Range:	$[0, \infty)$
Format:	Integer
Remark:	This is calculated by totaling the number of instance variables defined in the class and the number of methods defined in the class [21]. Smaller class sizes usually indicate a better designed system reflecting better distributed responsibilities. In other words, all of the functionality was not just stuffed into one big class. It is difficult to set hard and fast rules about this, but we should look carefully at classes where UWCS is above 100 [27].

Table A.23: Number of instance variables (or attributes) defined in this code module. (INST)

Entity:	Code Module(C,P)
Attribute:	Size
Unit:	Number of instance variables (INST)
Scale Type:	Ratio
Range:	$[0, \infty)$
Format:	Integer

Table A.24: Number of packages imported by this class (PACK)

Entity:	Code Module(C)
Attribute:	Complexity
Unit:	Number of package imported (PACK)
Scale Type:	Ratio
Range:	$[0, \infty)$
Format:	Integer

Table A.25: Response for class (RFC)

Entity:	Code Module(C)
Attribute:	Coupling
Unit:	$RFC = NOMT + EXT$
Scale Type:	Ratio
Range:	$[0, \infty)$
Format:	Integer
Remark	Response for class is calculated by totaling the number of methods declared in the class and the number of methods that are external to the class called from the code within the class. A high value for RFC indicates a class that is more complex and therefore more difficult to test and maintain [21]. Chidamber and Kemerer claim that [7]: *a*) If a large number of methods can be invoked in response to a message, the testing and debugging of the class becomes more complicated since it requires a greater level of understanding required on the part of the tester. *b*) The larger the number of methods that can be invoked from a class, the greater the complexity of the class.

Table A.26: Number of external method calls made from the class (EXT)

Entity:	Code Module(C)
Attribute:	Coupling
Unit:	Number of external method calls made from the class (EXT)
Scale Type:	Ratio
Range:	$[0, \infty)$
Format:	Integer
Remark :	Number of external method calls made from the class, i.e. the number of calls made to methods in other classes (including classes that are not in the group of classes under analysis, e.g. classes in the JDK, classes in third-party packages) [21].

Table A.27: Coupling Between Objects (CBO)

Entity:	Code Module(C,S)
Attribute:	Coupling
Unit:	Coupling Between Objects (CBO)
Scale Type:	Ratio
Range:	$[0, \infty)$
Format:	Integer
Remark :	Two sets of classes are created in order to calculate this measure: (1) classes in the code analyzed by JHawk that reference this class. (2) classes in the code analyzed by JHawk that this class references. From the intersection of these two sets a third set is created of classes that this class references and that reference this class. The size of this set is the value of the CBO measure [21]. At the system level these measures are cumulative for all classes in the system. Chidamber and Kemerer claim that [7]: *a*) Excessive coupling between object classes is detrimental to modular design and prevents reuse. The more independent a class is, the easier it is to reuse it in another application. *b*) In order to improve modularity and promote encapsulation, inter-object class couples should be kept to a minimum. The larger the number of couples, the higher the sensitivity to changes in other parts of the design, and therefore maintenance is more difficult. *c*) A measure of coupling is useful in order to determine how complex the testing of various parts of a design is likely to be. The higher the inter-object class coupling, the more rigorous the testing needs to be.

Table A.28: Total number of comment lines in the code module.(CCML)

Entity:	Code Module(C,S,P)
Attribute:	Size
Unit:	Total number of comment lines in the code module.(CCML)
Scale Type:	Ratio
Range:	$[0, \infty)$
Format:	Integer

Table A.29: The Distance measure.(DIST)

Entity:	Code Module(S,P)
Attribute:	Stability
Unit:	$DIST = abs(1 - (ABST + INST))$
Scale Type:	Ratio
Range:	$[0, \infty)$
Format:	Floating point
Remark :	A package should be balanced between abstractness and instability, i.e. somewhere between abstract and stable or concrete and unstable. Stable packages should also be abstract packages (A = 1 and I = 0) while unstable packages should be concrete (A = 0 and I = 1) [21]. At the system level we use a cumulative measure for all packages in the system. ABST is defined in [21].

Table A.30: Fan In (or Afferent Coupling).(FIN)

Entity:	Code Module(C,P,S)
Attribute:	Changeability.
Unit:	Fan In (or Afferent Coupling).(FIN)
Scale Type:	Ratio
Range:	$[0, \infty)$
Format:	Integer
Remark :	This is calculated as the number of modules in the code analyzed by JHawk that referenced this module [21].

Table A.31: Fan Out (or Efferent Coupling).(FOUT)

Entity:	Code Module(C,P,S)
Attribute:	Changeability.
Unit:	Fan Out (or Efferent Coupling).(FOUT)
Scale Type:	Ratio
Range:	$[0, \infty)$
Format:	Integer
Remark :	This is calculated as the number of modules in the code analyzed by JHawk that this class references [21].

Table A.32: Lack of Cohesion of Methods. (LCOM)

Entity:	Code Module(C,S)
Attribute:	Cohesion
Unit:	$LCOM = ((1/INST) * (numRefs - NOMT))/(1 - NOMT))$
Scale Type:	Ratio
Range:	$[0, 2]$
Format:	Floating point
Remark:	This is calculated according to the formula defined by Henderson-Sellars [19] where $NumRefs$ is the sum of the number of attribute references in each of the methods in the class. This version of LCOM has values in the range of 0 to 2. Lower values are better - any value over 1 should be viewed as an indicator a of poor code [21] Chidamber and Kemerer claim that low cohesion increases complexity, thereby increasing the likelihood of errors during the development process.

Table A.33: Lack of Cohesion of Methods. (LCOM2)

Entity:	Code Module(C)
Attribute:	Cohesion
Unit:	Lack of Cohesion of Methods. (LCOM2)
Scale Type:	Ratio
Range:	$[0, \infty)$
Format:	Floating point
Remark:	The LCOM2 measure is calculated by keeping a count of the number of method pairs that share instance variable references and a separate count of those that do not share instance variable references. The number of those that have instance variable references in common is subtracted from those that have none in common. If the value returned is negative, it is set to zero [5, 34].

Appendix B

Maintainability Analysis Model

Quality Evaluation Model

Results obtained from the characteristics and sub-characteristics are divided into ranges corresponding to the different degrees of satisfaction of the requirements. A satisfactory quality component can be rated as:

- *excellent* if it respects a all the rules defined by the evaluation model;
- *good* if it deviates slightly from the model;
- *fair* if many rules have been violated; and
- *poor* if it has a poor aptitude to provide efficient maintainability activity.

Oman's Maintainability Model(MI)

We describe the Maintainability Index model in Chapter 4.3 and the corresponding measure is presented in Table B.1. The evaluation model for Oman's Maintainability Index is presented in Table B.2.

Table B.1: Maintainability Index No Comments (MINC)

Entity:	Code Module(S)
Attribute:	Maintainability
Unit:	$MI = 171 - 3,42 * ln(HEFF/NOMT) - 0,23(TCC/NOMT) - 16,2 * ln(NOS/NOMT)$
Scale Type:	Ratio
Range:	$[0, 171]$
Format:	Floating point

McCall's Maintainability Model(MCC)

We describe this model in Chapter 4.3, and the five corresponding criteria are presented in the Tables B.3,B.4, B.5, B.6, B.7. We present the model called McCall's Maintainability in Table B.8.

Table B.2: Evaluation Model for Oman's Maintainability Index Model.

categories	min	*max*
Excellent	136	171
Good	126	135
Fair	116	125
Poor	0	115

Table B.3: Self-descriptiveness. (SELD)

Entity:	Code Module(C)
Attribute:	Self-descriptiveness. (SELD)
Unit:	$SELD = AVCCML$
Scale Type:	Ratio
Range:	$[0, 1]$
Format:	Integer
Remark:	A higher value of SELD means a source code which provides better documentation that explains implementation of the components.

Table B.4: Consistency for code module level. (CONS)

Entity:	Code Module(C)
Attribute:	Consistency for code module level. (CONS)
Unit:	$CONS = 0,7 * AVLCOM + 0,3 * AVUWCS$
Scale Type:	Ratio
Range:	$[0, 1]$
Format:	Floating point
Remark:	A lower value of CONS means a source code that uses a more uniform design as well as implementation techniques and notation.

Table B.5: Conciseness for code module level. (CONC)

Entity:	Code Module(C)
Attribute:	Conciseness for code module level. (CONC)
Unit:	$CONC = 0,9 * AVNOS + 0,1 * AVUWCS$
Scale Type:	Ratio
Range:	$[0, 1]$
Format:	floating point
Remark:	A lower value of CONC means a source code that is more compact in terms of the number of statements.

ISO/IEC 9126 Maintainability Model(ISO)

We describe this model in Chapter 4.3 and the four corresponding sub-characteristics are presented in Tables B.9, B.10, B.11, B.12. We present the model ISO/IEC 9126 Maintainability

Table B.6: Simplicity for code module level. (SIMP)

Entity:	Code Module(C)
Attribute:	Simplicity for code module level. (SIMP)
Unit:	$SIMP = 0,4 * AVUWCS + 0,3 * AVRFC + 0,3 * AVLCOM$
Scale Type:	Ratio
Range:	$[0, 1]$
Format:	Floating point
Remark:	A lower value of SIMP means a source code that can be better understood and tested.

Table B.7: Modularity for code module level. (MODU)

Entity:	Code Module(C)
Attribute:	Modularity for code module level. (MODU)
Unit:	$MODU = 0,4 * AVUWCS + 0,3 * AVCBO + 0,3 * AVEXT$
Scale Type:	Ratio
Range:	$[0, 1]$
Format:	Floating point
Remark:	A lower value of MODU means a source code with more independent modules.

Table B.8: McCall's Maintainability model for code module level. (MCC)

Entity:	Code Module(C)
Attribute:	Maintainability for code module level. (MCC)
Unit:	$MCC = 0,24 * CONS + 0,24 * CONC - 0,04 * SELD + 0,24 * SIMP + 0,24 * MODU$
Scale Type:	Ratio
Range:	$[0, 1]$
Format:	Floating point
Remark:	A lower value of MCC means that the effort required to locate and fix a defect in an operational program is smaller.

in Table B.13.

Table B.9: Analyzability for code module level. (ANAL)

Entity:	Code Module(C)
Attribute:	Analyzability for code module level. (ANAL)
Unit:	$ANAL = 0,4 * AVNOS + 0,4 * AVRFC + 0,2 * AVHEFF)$
Scale Type:	Ratio
Range:	$[0,1]$
Format:	Floating point
Remark:	A lower value of ANAL means that the effort required for diagnosis of failures or for identification of parts to be modified is smaller.

Table B.10: Changeability for code module level. (CHAN)

Entity:	Code Module(C)
Attribute:	Changeability for code module level. (CHAN)
Unit:	$CHAN = 0,3 * AVNOS + 0,3 * AVCBO + 0,3 * AVEXT + 0,1 * AVHEFF$
Scale Type:	Ratio
Range:	$[0,1]$
Format:	Floating point
Remark:	A lower value of CHAN means that the effort required for modification, fault removal or environment change is smaller.

Table B.11: Stability for code module level. (STAB)

Entity:	Code Module(C)
Attribute:	Stability for code module level. (STAB)
Unit:	$STAB = 0,3 * AVNOS + 0,3 * AVCBO + 0,1 * AVEXT + 0,1 * AVINST + 0,1 * VAPACK + 0,1 * AVLCOM$
Scale Type:	Ratio
Range:	$[0,1]$
Format:	Floating point
Remark:	A lower value of STAB means that the risk of unexpected effects of modification is smaller.

Table B.12: Testability for code module level. (TEST)

Entity:	Code Module(C)
Attribute:	Testability for code module level. (TEST)
Unit:	$TEST = 0,4 * AVNOS + 0,3 * AVCBO + 0,3 * AVRFC$
Scale Type:	Ratio
Range:	$[0, 1]$
Format:	Floating point
Remark:	A lower value of TEST means that the effort required for validating the modified software is smaller.

Table B.13: ISO/IEC 9126 Maintainability model for code module level. (ISO)

Entity:	Code Module(C)
Attribute:	Maintainability for code module level. (ISO)
Unit:	$ISO = 0,25 * ANAL + 0,25 * CHAN + 0,25 * STAB + 0,25 * TEST$
Scale Type:	Ratio
Range:	$[0, 1]$
Format:	Floating point
Remark:	A lower value of ISO means that the effort required to make a specified modification (which may include correcting, improving or adopting software to environment changes and modifications in the requirements and functional specification) is smaller.

Appendix C

Complexity Experiment Results

This Appendix presents the complexity experiment results for four case studies cases. These case studies includes the following design patterns:

- case 1, described in Section 6.2 - *Template View*, *Page Controller*;
- case 2, described in Section 6.3 - *Page Controller*;
- case 3, described in Section 6.4 - *Template View*, *Front Controller*;
- case 4, described in Section 6.5 - *Front Controller*.

For each of the four case studies, we present all measures of data for each of the code modules at three levels of examination, namely: *a*) *system*; *b*) *package*; and *c*) *class* . For the method level of examination, we do not present all measures of data, due to the very long list with all of the methods.

We also provide three code module levels of examination, namely: *a*) *package*; *b*) *class*; and *c*) *method* . As well as lowest value (LOW), highest value (HI) and average value (AVERAGE).

The name of the package is truncated and does not include the full name of the package. The truncated part of each package is *pl.arturkb.EInvoice*. We did this due to the space limit.

We provided the results in CSV, HTML, ODS format as well as all of the source codes for each of case study. Follow this link https://dl.dropboxusercontent.com/u/82268612/Experiment.zip to download all of the content.

Package level CASE1

System overview for CASE1

Name	**NLOC**	**AVCC**	TCC	**NOS**	HEFF	HBUG	CCML
CASE1	1063	1.36	165	807	279545.16	9.84	1113

AVLOC	**AVNOS**	**AVHEFF**	**AVHBUG**	**AVHLTH**	**AVHVOL**
6.52	4.95	1715.00	0.06	36.77	181.13

Package level

Name	NLOC	AVCC	TCC	No. Methods	NOS	HLTH	HEFF	HBUG	HVOL	CCML
.Beans.Alert	137	1	27	27	95	594	11021.03	0.7	2095.41	241
.Beans.Model	44	1.09	12	11	31	164	1940.22	0.19	555.97	52
.Controller	68	2.25	9	4	54	440	17209.8	0.74	2210.94	59
.Controller.Dashboard	51	1	4	4	43	358	12280.33	0.62	1847.99	44
.Controller.User	323	3.07	43	14	254	2113	162336.3	4.03	12090.57	134
.Filter	108	1.67	15	9	77	651	28404.59	1.06	3166.68	106
.Internationalization	104	1	2	2	92	652	22049.33	1.26	3794.91	22
.UI	182	1	46	46	128	765	15342.02	0.87	2600.64	399
.Utils	46	1.75	7	4	33	257	8961.55	0.39	1160.91	56
		13.83								

	NLOC	AVCC	TCC	No. Methods	NOS	HLTH	HEFF	HBUG	HVOL	CCML
HI	323	3.07	46	46	254	2113	162336.3	4.03	12090.57	399
LOW	44	1	2	2	31	164	1940.22	0.19	555.97	22
AVERAGE	118.11	1.54	18.33	13.44	89.67	666.00	31060.57	1.10	3280.45	123.67

No. Methods(NOMT)	121
No. Classes	33
No. Packages	9
	163

Class level CASE1

Package	Name	NLOC	AVCC	TCC	No. Methods	NOS	HEFF	HBUG	UWCS	INST	PACK	RFC	CBO	CCML	LCOM
.Beans.Alert	Alert	5	1	2	2	4	380.16	0.03	3	1	0	3	10	19	1
.Beans.Alert	ErrorAlert	14	1	2	2	9	1327.84	0.08	3	1	0	2	2	29	0
.Beans.Alert	FormEmptyAlert	21	1	5	5	13	932.2	0.08	7	2	0	5	2	22	0.38
.Beans.Alert	FormErrorAlert	25	1	5	5	17	1640.57	0.12	7	2	0	5	2	22	0
.Beans.Alert	FormNotChangedAlert	25	1	5	5	17	1640.57	0.12	7	2	0	5	2	22	0
.Beans.Alert	FormSuccessAlert	25	1	5	5	17	1640.57	0.12	7	2	0	5	2	22	0
.Beans.Alert	InfoAlert	15	1	3	3	11	1033.62	0.08	4	1	0	3	2	28	0
.Beans.Model	User	43	1.09	12	11	30	1890.72	0.17	16	5	0	11	4	49	0
.Controller	ChangeLanguage	25	2	6	3	16	4550.84	0.22	5	2	9	4	2	31	0.5
.Controller	Log4jInit	25	3	3	1	20	8815.96	0.26	2	1	7	1	0	25	1
.Controller.Dashboard	Index	36	1	4	4	28	8921.75	0.38	5	1	14	4	7	44	1
.Controller.User	Edit	156	3.86	27	7	120	78868.39	1.85	9	2	23	8	15	55	0.25
.Controller.User	Login	96	3	12	4	70	49091.29	1.08	6	2	19	5	11	50	0.33
.Controller.User	Logout	19	1.33	4	3	12	2169.83	0.13	4	1	7	3	1	29	1
.Filter	InternationalizationFilter	24	1.33	4	3	15	4329.72	0.2	5	2	12	4	1	38	0.25
.Filter	InternationalizationFilterSecure	20	1.33	4	3	11	3375.44	0.16	4	1	10	3	1	38	1
.Filter	LoginFilter	28	2.33	7	3	15	8075.23	0.22	3	0	11	3	1	21	0
.Internationalization	EnglishLang	49	1	1	1	44	11291.81	0.61	2	1	1	2	3	13	0
.Internationalization	PolishLang	51	1	1	1	44	10314.71	0.62	2	1	1	2	1	6	0
.UI	Body	23	1	6	6	16	916.94	0.09	9	3	0	6	8	31	0
.UI	Breadcrumb	12	1	3	3	8	789.8	0.06	4	1	2	3	4	10	0
.UI	BreadcrumbItem	16	1	4	4	11	636.45	0.06	6	2	0	4	3	22	0
.UI	Content	4	1	2	2	3	58.89	0.01	2	0	0	2	4	14	0
.UI	DashboardContent	14	1	3	3	8	617.49	0.05	4	1	0	3	2	7	0
.UI	EditContent	25	1	6	6	16	1212.07	0.1	9	3	1	6	3	18	0
.UI	HTML5	8	1	6	6	7	152.68	0.03	6	0	0	6	3	28	0
.UI	Head	9	1	2	2	6	316.38	0.03	3	1	0	2	5	13	0
.UI	LoginContent	18	1	4	4	11	868.83	0.07	6	2	1	4	3	19	0
.UI	NavBarTop	9	1	2	2	6	358.51	0.04	3	1	0	2	1	13	0
.UI	NavigationBar	4	1	2	2	3	58.89	0.01	2	0	0	2	2	13	0
.UI	Page	23	1	6	6	16	986.52	0.1	9	3	0	6	6	45	0
.Utils	HibernateUtil	15	1.5	3	2	9	1700.46	0.09	3	1	2	2	2	21	1
.Utils	ServletsUtils	22	2	4	2	15	5758.07	0.18	3	1	5	3	3	29	0

	NLOC	AVCC	TCC	No. M	NOS	HEFF	HBUG	**UWCS**	**INST**	**PACK**	**RFC**	**CBO**	**CCML**	**LCOM**
HI	156.00	3.86	27.00	11.00	120.00	78868.39	1.85	16.00	5.00	23.00	11.00	15.00	55.00	1.00
LOW	4.00	1.00	1.00	1.00	3.00	58.89	0.01	2.00	0.00	0.00	1.00	0.00	6.00	0.00
AVERAGE per class	27.39	1.36	5.00	3.67	19.64	6506.76	0.23	5.15	1.48	3.79	3.91	3.58	25.64	0.23

Class level CASE1

Package	Name	LCOM2	EXT
.Beans.Alert	Alert	2	1
.Beans.Alert	ErrorAlert	2	0
.Beans.Alert	FormEmptyAlert	5	0
.Beans.Alert	FormErrorAlert	11	0
.Beans.Alert	FormNotChangedAlert	11	0
.Beans.Alert	FormSuccessAlert	11	0
.Beans.Alert	InfoAlert	1	0
.Beans.Model	User	61	0
.Controller	ChangeLanguage	4	1
.Controller	Log4jInit	1	0
.Controller.Dashboard	Index	4	0
.Controller.User	Edit	3	1
.Controller.User	Login	4	1
.Controller.User	Logout	3	0
.Filter	InternationalizationFilter	4	1
.Filter	InternationalizationFilterSecure	3	0
.Filter	LoginFilter	3	0
.Internationalization	EnglishLang	0	1
.Internationalization	PolishLang	0	1
.UI	Body	18	0
.UI	Breadcrumb	1	0
.UI	BreadcrumbItem	8	0
.UI	Content	2	0
.UI	DashboardContent	1	0
.UI	EditContent	18	0
.UI	HTML5	6	0
.UI	Head	2	0
.UI	LoginContent	8	0
.UI	NavBarTop	2	0
.UI	NavigationBar	2	0
.UI	Page	18	0
.Utils	HibernateUtil	2	0
.Utils	ServletsUtils	0	1

	LCOM2	**EXT**
HI	61.00	1.00
LOW	0.00	0.00
AVARAGE	6.70	0.24

Method level CASE1

	NLOC	COMP	NOCL	NOS	HLTH	NAND	NOPR	HVOC	HEFF	HDIF	HBUG	CREF	XMET	LMET
HI	57	7	17	50	399	184	215	94	42106.02	18.59	0.87	21	34	4
LOW	1	1	0	1	5	2	3	5	17.41	1.5	0	0	1.5	0
AVERAGE	5.74	1.36	3.55	4.02	28.50	13.52	14.98	15.17	1534.13	4.55	0.05	2.14	1.68	0.06

Package level CASE2

System overview for CASE2

Name	**NLOC**	**AVCC**	TCC	**NOS**	HEFF	HBUG	CCML
CASE2	1181	1.8	106	997	502822.21	13.86	550

AVLOC	**AVNOS**	**AVHEFF**	**AVHBUG**	**AVHLTH**	**AVHVOL**
14.76	12.46	6285.28	0.17	91.90	519.58

Package level

Name	NLOC	AVCC	TCC	No. Methods	NOS	HLTH	HEFF	HBUG	HVOL	CCML
.Beans.Model	44	1.09	12	11	31	164	1940.22	0.19	555.97	52
.Controller	68	2.25	9	4	54	440	17209.8	0.74	2210.94	59
.Filter	108	1.67	15	9	77	651	28404.59	1.06	3166.68	106
.Internationalization	104	1	2	2	92	652	22049.33	1.26	3794.91	22
.Servlet	426	2.35	54	23	333	2664	184077.53	4.78	14340.78	214
.Utils	431	1.4	14	10	410	2781	249140.73	5.83	17497.02	97

	NLOC	AVCC	TCC	No. Methods	NOS	HLTH	HEFF	HBUG	HVOL	CCML
HI	431	2.35	54	23	410	2781	249140.73	5.83	17497.02	214
LOW	44	1	2	2	31	164	1940.22	0.19	555.97	22
AVERAGE	196.83	1.63	17.67	9.83	166.17	1225.33	83803.70	2.31	6927.72	91.67

No. Methods(NOMT)	59
No. Classes	15
No. Packages	6
	80

Class level CASE2

Package	Name	NLOC	AVCC	TCC	No. Methods	NOS	HEFF	HBUG	UWCS	INST	PACK	RFC	CBO	CCML	LCOM
.Beans.Model	User	43	1.09	12	11	30	1890.72	0.17	16	5	0	11	6	49	0
.Controller	ChangeLanguage	25	2	6	3	16	4550.84	0.22	5	2	9	4	2	31	0.5
.Controller	Log4jInit	25	3	3	1	20	8815.96	0.26	2	1	7	1	0	25	1
.Filter	InternationalizationFilter	24	1.33	4	3	15	4329.72	0.2	5	2	12	4	1	38	0.25
.Filter	InternationalizationFilterSecure	20	1.33	4	3	11	3375.44	0.16	4	1	10	3	1	38	1
.Filter	LoginFilter	28	2.33	7	3	15	8075.23	0.22	3	0	11	3	1	21	0
.Internationalization	EnglishLang	49	1	1	1	44	11291.81	0.61	2	1	1	2	3	13	0
.Internationalization	PolishLang	51	1	1	1	44	10314.71	0.62	2	1	1	2	1	6	0
.Servlet	Dashboard_Index	31	1.67	5	3	22	7908.52	0.28	5	2	11	4	4	38	2
.Servlet	Login	167	2.14	15	7	131	88752.01	2.03	9	2	13	8	6	50	0.5
.Servlet	Logout	19	1.33	4	3	12	2169.83	0.13	4	1	7	3	1	29	1
.Servlet	User_Edit	160	3	30	10	119	54594.66	1.59	16	6	13	11	5	58	0.09
.Utils	HibernateUtil	15	1.5	3	2	9	1700.46	0.09	3	1	2	2	2	21	1
.Utils	ServletsUtils	31	2	6	3	22	7830.96	0.25	4	1	5	4	5	34	0
.Utils	UI	371	1	5	5	365	236592.65	5.29	5	0	4	5	6	36	0

	NLOC	AVCC	TCC	No. Methods	NOS	HEFF	HBUG	**UWCS**	**INST**	**PACK**	**RFC**	**CBO**	**CCML**	**LCOM**
HI	371.00	3.00	30.00	11.00	365.00	236592.65	5.29	16.00	6.00	13.00	11.00	6.00	58.00	2.00
LOW	15.00	1.00	1.00	1.00	9.00	1700.46	0.09	0.00	0.00	0.00	1.00	0.00	6.00	0.00
AVERAGE	70.60	1.71	7.07	3.93	58.33	30146.23	0.81	5.67	1.73	7.07	4.47	2.93	32.47	0.49

Class level CASE2

Package	Name	LCOM2	EXT
.Beans.Model	User	61	0
.Controller	ChangeLanguage	4	1
.Controller	Log4jInit	1	0
.Filter	InternationalizationFilter	4	1
.Filter	InternationalizationFilterSecure	3	0
.Filter	LoginFilter	3	0
.Internationalization	EnglishLang	0	1
.Internationalization	PolishLang	0	1
.Servlet	Dashboard_Index	3	1
.Servlet	Login	12	1
.Servlet	Logout	3	0
.Servlet	User_Edit	34	1
.Utils	HibernateUtil	2	0
.Utils	ServletsUtils	0	1
.Utils	UI	5	0

	LCOM2	**EXT**
HI	61.00	1.00
LOW	0.00	0.00
AVARAGE	9.00	0.53

Method level CASE2

	NLOC	COMP	NOCL	NOS	HLTH	NAND	NOPR	HVOC	HEFF	HDIF	HBUG	CREF	XMET	LMET
HI	117	7	12	116	827	417	410	141	104167.4	18.77	1.97	14	30	6
LOW	2	1	0	1	7	2	4	6	36.19	2	0.01	0	0	0
AVERAGE	15.42	1.80	4.53	12.78	91.92	44.90	47.02	29.12	7195.06	7.40	0.18	3.61	3.44	0.17

Package level CASE3

System overview for CASE3

Name	**NLOC**	**AVCC**	TCC	**NOS**	HEFF	HBUG	CCML
CASE3	1309	1.44	207	993	339179.45	11.76	1082

AVLOC	**AVNOS**	**AVHEFF**	**AVHBUG**	**AVHLTH**	**AVHVOL**
6.82	5.17	192.00	0.06	1766.56	183.68

Package level

Name	NLOC	AVCC	TCC	No. Methods	NOS	HLTH	HEFF	HBUG	HVOL	CCML
.Beans.Alert	137	1	27	27	95	594	11021.03	0.7	2095.41	241
.Beans.Model	44	1.09	12	11	31	164	1940.22	0.19	555.97	52
.Controller	33	3	3	1	28	209	9669.25	0.36	1068.71	28
.Filter	108	1.67	15	9	77	651	28404.59	1.06	3166.68	106
.FrontController	255	2.06	37	18	196	1523	83965.6	2.65	7942.9	62
.FrontControllerSecure	400	2.23	58	26	313	2421	157825.87	4.29	12880.66	116
.Internationalization	104	1	2	2	92	652	22049.33	1.26	3794.91	22
.UI	182	1	46	46	128	765	15342.02	0.87	2600.64	399
.Utils	46	1.75	7	4	33	257	8961.55	0.39	1160.91	56

	NLOC	AVCC	TCC	No. Methods	NOS	HLTH	HEFF	HBUG	HVOL	CCML
HI	400	3	58	46	313	2421	157825.87	4.29	12880.66	399
LOW	33	1	2	1	28	164	1940.22	0.19	555.97	22
AVERAGE	145.44	1.64	23.00	16.00	110.33	804.00	37686.61	1.31	3918.53	120.22

No. Methods(NOMT)	144
No. Classes	39
No. Packages	9
	192

Class level CASE3

Package	Name	NLOC	AVCC	TCC	No. Methods	NOS	HEFF	HBUG	UWCS	INST	PACK	RFC	CBO	CCML	LCOM
.Beans.Alert	Alert	5	1	2	2	4	380.16	0.03	3	1	0	3	10	19	1
.Beans.Alert	ErrorAlert	14	1	2	2	9	1327.84	0.08	3	1	0	2	2	29	0
.Beans.Alert	FormEmptyAlert	21	1	5	5	13	932.2	0.08	7	2	0	5	2	22	0.38
.Beans.Alert	FormErrorAlert	25	1	5	5	17	1640.57	0.12	7	2	0	5	2	22	0
.Beans.Alert	FormNotChangedAlert	25	1	5	5	17	1640.57	0.12	7	2	0	5	2	22	0
.Beans.Alert	FormSuccessAlert	25	1	5	5	17	1640.57	0.12	7	2	0	5	2	22	0
.Beans.Alert	InfoAlert	15	1	3	3	11	1033.62	0.08	4	1	0	3	2	28	0
.Beans.Model	User	43	1.09	12	11	30	1890.72	0.17	16	5	0	11	4	49	0
.Controller	Log4jInit	25	3	3	1	20	8815.96	0.26	2	1	7	1	0	25	1
.Filter	InternationalizationFilter	24	1.33	4	3	15	4329.72	0.2	5	2	12	4	1	38	0.25
.Filter	InternationalizationFilterSecure	20	1.33	4	3	11	3375.44	0.16	4	1	10	3	1	38	1
.Filter	LoginFilter	28	2.33	7	3	15	8075.23	0.22	3	0	11	3	1	21	0
.FrontController	ChangeLanguageCommand	33	2.25	9	4	23	5348.79	0.27	6	2	9	5	3	29	0.33
.FrontController	FrontCommand	19	1	3	3	16	1933.58	0.14	8	5	6	3	4	0	0.5
.FrontController	FrontServlet	55	1.67	10	6	36	8608.99	0.4	8	2	6	7	2	5	0.5
.FrontController	LoginCommand	98	3.5	14	4	73	47619.97	1.07	5	1	17	5	11	28	0.33
.FrontController	UnknowCommand	5	1	1	1	3	134.8	0.02	1	0	2	1	2	0	0
.FrontControllerSecure	DashBoardIndexCommand	45	1.4	7	5	37	10108.01	0.41	7	2	13	6	8	44	0.5
.FrontControllerSecure	EditCommand	201	3.78	34	9	151	97954.56	2.17	11	2	23	10	15	67	0.19
.FrontControllerSecure	FrontCommand	19	1	3	3	16	1933.58	0.14	8	5	6	3	5	0	0.5
.FrontControllerSecure	FrontServlet	55	1.67	10	6	36	8608.99	0.4	8	2	6	7	2	5	0.5
.FrontControllerSecure	LogoutCommand	15	1.5	3	2	10	1982.83	0.1	2	0	4	2	2	0	0
.FrontControllerSecure	UnknowCommand	5	1	1	1	3	134.8	0.02	1	0	2	1	2	0	0
.Internationalization	EnglishLang	49	1	1	1	44	11291.81	0.61	2	1	1	2	3	13	0
.Internationalization	PolishLang	51	1	1	1	44	10314.71	0.62	2	1	1	2	1	6	0
.UI	Body	23	1	6	6	16	916.94	0.09	9	3	0	6	8	31	0
.UI	Breadcrumb	12	1	3	3	8	789.8	0.06	4	1	2	3	4	10	0
.UI	BreadcrumbItem	16	1	4	4	11	636.45	0.06	6	2	0	4	3	22	0
.UI	Content	4	1	2	2	3	58.89	0.01	2	0	0	2	4	14	0
.UI	DashboardContent	14	1	3	3	8	617.49	0.05	4	1	0	3	2	7	0
.UI	EditContent	25	1	6	6	16	1212.07	0.1	9	3	1	6	3	18	0
.UI	HTML5	8	1	6	6	7	152.68	0.03	6	0	0	6	3	28	0
.UI	Head	9	1	2	2	6	316.38	0.03	3	1	0	2	5	13	0
.UI	LoginContent	18	1	4	4	11	868.83	0.07	6	2	1	4	3	19	0
.UI	NavBarTop	9	1	2	2	6	358.51	0.04	3	1	0	2	1	13	0
.UI	NavigationBar	4	1	2	2	3	58.89	0.01	2	0	0	2	2	13	0
.UI	Page	23	1	6	6	16	986.52	0.1	9	3	0	6	6	45	0
.Utils	HibernateUtil	15	1.5	3	2	9	1700.46	0.09	3	1	2	2	2	21	1
.Utils	ServletsUtils	22	2	4	2	15	5758.07	0.18	3	1	5	3	3	29	0

	NLOC	AVCC	TCC	No. Methods	NOS	HEFF	HBUG	**UWCS**	**INST**	**PACK**	**RFC**	**CBO**	**CCML**	**LCOM**
HI	201.00	3.78	34.00	11.00	151.00	97954.56	2.17	16.00	5.00	23.00	11.00	15.00	67.00	1.00
LOW	4.00	1.00	1.00	1.00	3.00	58.89	0.01	1.00	0.00	0.00	1.00	0.00	0.00	0.00
AVERAGE	28.77	1.37	5.31	3.69	20.67	6551.03	0.23	5.28	1.59	3.77	3.97	3.54	20.90	0.20

Class level CASE3

Package	Name	LCOM2	EXT
.Beans.Alert	Alert	2	1
.Beans.Alert	ErrorAlert	2	0
.Beans.Alert	FormEmptyAlert	5	0
.Beans.Alert	FormErrorAlert	11	0
.Beans.Alert	FormNotChangedAlert	11	0
.Beans.Alert	FormSuccessAlert	11	0
.Beans.Alert	InfoAlert	1	0
.Beans.Model	User	61	0
.Controller	Log4jInit	1	0
.Filter	InternationalizationFilter	4	1
.Filter	InternationalizationFilterSecure	3	0
.Filter	LoginFilter	3	0
.FrontController	ChangeLanguageCommand	4	1
.FrontController	FrontCommand	4	0
.FrontController	FrontServlet	10	1
.FrontController	LoginCommand	0	1
.FrontController	UnknowCommand	1	0
.FrontControllerSecure	DashBoardIndexCommand	10	1
.FrontControllerSecure	EditCommand	0	1
.FrontControllerSecure	FrontCommand	4	0
.FrontControllerSecure	FrontServlet	10	1
.FrontControllerSecure	LogoutCommand	2	0
.FrontControllerSecure	UnknowCommand	1	0
.Internationalization	EnglishLang	0	1
.Internationalization	PolishLang	0	1
.UI	Body	18	0
.UI	Breadcrumb	1	0
.UI	BreadcrumbItem	8	0
.UI	Content	2	0
.UI	DashboardContent	1	0
.UI	EditContent	18	0
.UI	HTML5	6	0
.UI	Head	2	0
.UI	LoginContent	8	0
.UI	NavBarTop	2	0
.UI	NavigationBar	2	0
.UI	Page	18	0
.Utils	HibernateUtil	2	0
.Utils	ServletsUtils	0	1

	LCOM2	**EXT**
HI	61.00	1.00
LOW	0.00	0.00
AVARAGE	6.38	0.28

Method level CASE3

	NLOC	COMP	NOCL	NOS	HLTH	NAND	NOPR	HVOC	HEFF	HDIF	HBUG	CREF	XMET	LMET
HI	52	7	15	46	346	155	191	85	41197.91	19.06	0.72	16	27	4
LOW	1	1	0	1	5	2	3	5	17.41	1.5	0	0	0	0
AVERAGE	6.17	1.44	2.99	4.34	30.22	14.27	15.95	16.13	1553.50	4.83	0.05	2.26	1.65	0.23

Package level CASE4

System overview for CASE4

Name	**NLOC**	**AVCC**	TCC	**NOS**	HEFF	HBUG	CCML
CASE4	1387	1.77	145	1154	523364.5	15.35	490

AVLOC	**AVNOS**	**AVHEFF**	**AVHBUG**	**AVHLTH**	**AVHVOL**
12.61	10.49	4757.86	0.14	4757.86	418.50

Package level

Name	NLOC	AVCC	TCC	No. Methods	NOS	HLTH	HEFF	HBUG	HVOL	CCML
.Beans.Model	44	1.09	12	11	31	164	1940.22	0.19	555.97	52
.Controller	33	3	3	1	28	209	9669.25	0.36	1068.71	28
.Filter	108	1.67	15	9	77	651	28404.59	1.06	3166.68	106
.FrontController	321	1.9	40	21	251	1952	119348.24	3.45	10362.74	82
.FrontControllerSecure	345	2.11	59	28	264	1941	91958.48	3.18	9537.99	103
.Internationalization	104	1	2	2	92	652	22049.33	1.26	3794.91	22
.Utils	432	1.4	14	10	411	2787	249994.39	5.85	17548.35	97

	NLOC	AVCC	TCC	No. Methods	NOS	HLTH	HEFF	HBUG	HVOL	CCML
HI	432	3	59	28	411	2787	249994.39	5.85	17548.35	106
LOW	33	1	2	1	28	164	1940.22	0.19	555.97	22
AVERAGE	198.14	1.74	20.71	11.71	164.86	1193.71	74766.36	2.19	6576.48	70.00

No. Methods(NOMT)	82
No. Classes	21
No. Packages	7
	110

Class level CASE4

Package	Name	NLOC	AVCC	TCC	No. Methods	NOS	HEFF	HBUG	UWCS	INST	PACK	RFC	CBO	CCML	LCOM
.Beans.Model	User	43	1.09	12	11	30	1890.72	0.17	16	5	0	11	6	49	0
.Controller	Log4jInit	25	3	3	1	20	8815.96	0.26	2	1	7	1	0	25	1
.Filter	InternationalizationFilter	24	1.33	4	3	15	4329.72	0.2	5	2	12	4	1	38	0.25
.Filter	InternationalizationFilterSecure	20	1.33	4	3	11	3375.44	0.16	4	1	10	3	1	38	1
.Filter	LoginFilter	28	2.33	7	3	15	8075.23	0.22	3	0	11	3	1	21	0
.FrontController	ChangeLanguageCommand	32	2.25	9	4	22	5048.83	0.26	5	1	9	5	3	29	0.67
.FrontController	FrontCommand	19	1	3	3	16	1933.58	0.14	8	5	6	3	4	0	0.5
.FrontController	FrontServlet	55	1.67	10	6	36	8608.99	0.4	8	2	6	7	2	5	0.5
.FrontController	LoginCommand	172	2.43	17	7	136	88940.18	2.06	8	1	10	8	5	48	0.83
.FrontController	UnknowCommand	5	1	1	1	3	134.8	0.02	1	0	2	1	2	0	0
.FrontControllerSecure	DashBoardIndexCommand	41	1.8	9	5	30	8235.96	0.31	6	1	8	6	3	40	1
.FrontControllerSecure	EditCommand	167	3	33	11	126	51158.37	1.57	16	5	11	12	5	58	0.1
.FrontControllerSecure	FrontCommand	19	1	3	3	16	1933.58	0.14	8	5	6	3	5	0	0.5
.FrontControllerSecure	FrontServlet	55	1.67	10	6	36	8608.99	0.4	8	2	6	7	2	5	0.5
.FrontControllerSecure	LogoutCommand	15	1.5	3	2	10	1982.83	0.1	2	0	4	2	2	0	0
.FrontControllerSecure	UnknowCommand	5	1	1	1	3	134.8	0.02	1	0	2	1	2	0	0
.Internationalization	EnglishLang	49	1	1	1	44	11291.81	0.61	2	1	1	2	3	13	0
.Internationalization	PolishLang	51	1	1	1	44	10314.71	0.62	2	1	1	2	1	6	0
.Utils	HibernateUtil	15	1.5	3	2	9	1700.46	0.09	3	1	2	2	2	21	1
.Utils	ServletsUtils	31	2	6	3	22	7830.96	0.25	4	1	5	4	5	34	0
.Utils	UI	372	1	5	5	366	237446.3	5.31	5	0	4	5	6	36	0

	NLOC	AVCC	TCC	No. Methods	NOS	HEFF	HBUG	**UWCS**	**INST**	**PACK**	**RFC**	**CBO**	**CCML**	**LCOM**
HI	372.00	3.00	33.00	11.00	366.00	237446.30	5.31	16.00	5.00	12.00	12.00	6.00	58.00	1.00
LOW	5.00	1.00	1.00	1.00	3.00	134.80	0.02	1.00	0.00	0.00	1.00	0.00	0.00	0.00
AVERAGE	59.19	1.61	6.90	3.90	48.10	22466.30	0.63	5.57	1.67	5.86	4.38	2.90	22.19	0.37

Class level CASE4

Package	Name	LCOM2	EXT
.Beans.Model	User	61	0
.Controller	Log4jInit	1	0
.Filter	InternationalizationFilter	4	1
.Filter	InternationalizationFilterSecure	3	0
.Filter	LoginFilter	3	0
.FrontController	ChangeLanguageCommand	4	1
.FrontController	FrontCommand	4	0
.FrontController	FrontServlet	10	1
.FrontController	LoginCommand	14	1
.FrontController	UnknowCommand	1	0
.FrontControllerSecure	DashBoardIndexCommand	14	1
.FrontControllerSecure	EditCommand	0	1
.FrontControllerSecure	FrontCommand	4	0
.FrontControllerSecure	FrontServlet	10	1
.FrontControllerSecure	LogoutCommand	2	0
.FrontControllerSecure	UnknowCommand	1	0
.Internationalization	EnglishLang	0	1
.Internationalization	PolishLang	0	1
.Utils	HibernateUtil	2	0
.Utils	ServletsUtils	0	1
.Utils	UI	5	0

	LCOM2	**EXT**
HI	61.00	1.00
LOW	0.00	0.00
AVARAGE	6.81	0.48

Method level CASE4

	NLOC	COMP	NOCL	NOS	HLTH	NAND	NOPR	HVOC	HEFF	HDIF	HBUG	CREF	XMET	LMET
HI	118	7	12	117	833	420	413	142	105021.05	18.79	1.99	12	30	6
LOW	1	1	0	1	7	2	4	6	36.19	2	0.01	0	0	0
AVERAGE	13.09	1.77	3.24	10.66	75.37	36.50	38.87	26.21	5401.43	6.81	0.14	3.27	2.84	0.37

Appendix D

Maintainability Experiment Results

In this Appendix we present the maintainability experiment results for four identical case studies, just as in Appendix C, and for three maintainability models. The three Maintainability models which we used for evaluation are precisely described in Chapter 4.3:

- MINC - in this research project we use the maintainability index which is provided by the JHawk tool and based on Oman's Models. We only take into evaluation the maintainability index without comments. We use MINC because we have not yet found a way to automatically assess the quality of comments;
- MCC - McCall's Quality Model. According to this model the maintainability factor can be measured by combining five criteria:
  - CONS - consistency;
  - CONC - conciseness;
  - SELD - self-descriptiveness;
  - SIMP - simplicity; and
  - MODU - modularity.
- ISO - ISO/IEC 9126 Model The International standard ISO/IEC 9126-3 defines maintainability as a set of four sub-characteristics:
  - ANAL - analyzability;
  - CHAN - changeability;
  - STAB - stability; and
  - TEST - testability.

We also presents the base measures used to calculate the criteria and sub-characteristics.

System level CASE1

System overview for CASE1

Name

CASE1

Oman's Model				MINC					
					115.52 Poor	higher = better			
						Excellent	136	171	
						Good	126	135	
						Fair	116	125	
						Poor	0	115	
McCall's Model			MCC						
			0.17			Lower = better			
	24.00%	24.00%	4.00%	24.00%	24.00%	Final weight			
	CONS	CONC	SELD	SIMP	MODU				
	0.18	0.08	0.38	0.26	0.27				
	lower = better	lower = better	higher = better	lower = better	lower = better				
ISO/IEC 9126			ISO						
			0.17			Lower = better			
	25.00%	25.00%	25.00%	25.00%		Final weight			
	ANAL	CHAN	STAB	TEST					
	0.16	0.16	0.16	0.19					
	lower = better	lower = better	lower = better	lower = better					

Base measures	AVUWCS	AVINST	AVPACK	AVRFC	AVCBO	AVCCML	AVLCOM	AVEXT	AVNOS	AVHEFF	AVBUG
MIN	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0	0.00	0
MAX	16.00	6.00	23.00	12.00	15.00	67.00	2.00	1.00	366	237466.00	5.31
AV	5.15	1.48	3.79	3.91	3.58	25.64	0.23	0.24	19.64	6506.76	0.23
Normalized AV	0.321875	0.246667	0.164783	0.325833	0.238667	0.382687	0.115000	0.240000	0.053661	0.027401	0.043315

System level CASE2

System overview for CASE2

Name

CASE2

Oman's Model				MINC					
					95.00 Poor	higher = better			
						Excellent	136	171	
						Good	126	135	
						Fair	116	125	
						Poor	0	115	

McCall's Model			MCC			
			0.25			Lower = better
24.00%	24.00%	4.00%	24.00%	24.00%	Final weight	
CONS	CONC	SELD	SIMP	MODU		
0.28	0.18	0.48	0.33	0.36		
lower = better	lower = better	higher = better	lower = better	lower = better		

ISO/IEC 9126		ISO			
		0.25		Lower = better	
25.00%	25.00%	25.00%	25.00%	Final weight	
ANAL	CHAN	STAB	TEST		
0.24	0.28	0.24	0.23		
lower = better	lower = better	lower = better	lower = better		

Base measures	AVUWCS	AVINST	AVPACK	AVRFC	AVCBO	AVCCML	AVLCOM	AVEXT	AVNOS	AVHEFF	AVBUG
MIN	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0	0.00	0
MAX	16.00	6.00	23.00	12.00	15.00	67.00	2.00	1.00	366	237466.00	5.31
AV	5.67	1.73	7.07	4.47	2.93	32.47	0.49	0.53	58.33	30146.23	0.81
Normalized AV	0.354375	0.2883333333	0.3073913043	0.3725	0.1953333333	0.4846268657	0.245	0.53	0.1594	0.1269	0.1522

System level CASE3

System overview for CASE3

Name

CASE3

Oman's Model			MINC					
					115.49 Poor	higher = better		
						Excellent	136	171
						Good	126	135
						Fair	116	125
						Poor	0	115

McCall's Model			MCC			
			0.18			Lower = better
	24.00%	24.00%	4.00%	24.00%	24.00%	Final weight
	CONS	CONC	SELD	SIMP	MODU	
	0.17	0.08	0.31	0.26	0.29	
	lower = better	lower = better	higher = better	lower = better	lower = better	

ISO/IEC 9126			ISO		
			0.17		Lower = better
	25.00%	25.00%	25.00%	25.00%	Final weight
	ANAL	CHAN	STAB	TEST	
	0.16	0.17	0.17	0.19	
	lower = better	lower = better	lower = better	lower = better	

Base measures	AVUWCS	AVINST	AVPACK	AVRFC	AVCBO	AVCCML	AVLCOM	AVEXT	AVNOS	AVHEFF	AVBUG
MIN	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0	0.00	0
MAX	16.00	6.00	23.00	12.00	15.00	67.00	2.00	1.00	366	237466.00	5.31
AV	5.28	1.59	3.77	3.97	3.54	20.90	0.20	0.28	20.67	6551.03	0.23
Normalized AV	0.330000	0.265000	0.163913	0.330833	0.236000	0.311940	0.100000	0.280000	0.056475	0.027587	0.043315

System level CASE4

System overview for CASE4

Name

CASE4

Oman's Model				MINC				
					98.22 Poor	higher = better		
						Excellent	136	171
						Good	126	135
						Fair	116	125
						Poor	0	115

McCall's Model			MCC			
			0.23			Lower = better
24.00%	24.00%	4.00%	24.00%	24.00%	Final weight	
CONS	CONC	SELD	SIMP	MODU		
0.23	0.15	0.33	0.30	0.34		
lower = better	lower = better	higher = better	lower = better	lower = better		

ISO/IEC 9126		ISO			
		0.23		Lower = better	
25.00%	25.00%	25.00%	25.00%	Final weight	
ANAL	CHAN	STAB	TEST		
0.22	0.25	0.22	0.22		
lower = better	lower = better	lower = better	lower = better		

Base measures	AVUWCS	AVINST	AVPACK	AVRFC	AVCBO	AVCCML	AVLCOM	AVEXT	AVNOS	AVHEFF	AVBUG
MIN	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0	0.00	0
MAX	16.00	6.00	23.00	12.00	15.00	67.00	2.00	1.00	366	237466.00	5.31
AV	5.57	1.67	5.86	4.38	2.90	22.19	0.37	0.48	48.10	22466.30	0.63
Normalized AV	0.348125	0.278333	0.254783	0.365000	0.193333	0.331194	0.185000	0.480000	0.131421	0.094608	0.118644

Bibliography

[1] Deepak Alur, John Crupi and Dan Malks. *Core J2EE Patterns: Best Practices and Design Strategies*. Pearson Education, 2001. ISBN: 0130648841.

[2] Apache. *Apache Tomcat*. http://tomcat.apache.org/. Online; accessed 25-Julli-2013.

[3] Yirsaw Ayalew and Kagiso Mguni. 'An Assessment of Changeability of Open Source Software'. In: *Computer and Information Science* 6.3 (2013), p68.

[4] Alexandre Bergel et al. 'SQUALE - Software QUALity Enhancement'. In: *Proceedings of the 2009 European Conference on Software Maintenance and Reengineering*. CSMR '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 285–288. ISBN: 978-0-7695-3589-0. URL: http://dx.doi.org/10.1109/CSMR.2009.13.

[5] Lionel C Briand, John W Daly and Jürgen Wüst. 'A unified framework for cohesion measurement in object-oriented systems'. In: *Empirical Software Engineering* 3.1 (1998), pp. 65–117.

[6] S.R. Chidamber and C.F. Kemerer. 'A metrics suite for object oriented design'. In: *Software Engineering, IEEE Transactions on* 20.6 (1994), pp. 476–493. ISSN: 0098-5589.

[7] S.R. Chidamber and C.F. Kemerer. 'A metrics suite for object oriented design'. In: *Software Engineering, IEEE Transactions on* 20.6 (1994), pp. 476–493. ISSN: 0098-5589. DOI: 10.1109/32.295895.

[8] Don Coleman, Bruce Lowther and Paul Oman. 'The application of software maintainability models in industrial software systems'. In: *Journal of Systems and Software* 29.1 (1995), pp. 3–16.

[9] JBoss Community. *Hibernate*. http://www.hibernate.org/. Online; accessed 25-Julli-2013.

[10] Evans. *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN: 0321125215.

[11] Norman E. Fenton and Shari Lawrence Pfleeger. *Software metrics : a rigorous and practical approach / N.E. Fenton, S.L. Pfleeger.* 1997.

[12] The Document Foundation. *LibreOffice Calc*. http://www.libreoffice.org/features/calc/. Online; accessed 15-December-2013.

[13] The Eclipse Foundation. *Eclipse*. http://www.eclipse.org. Online; accessed 15-December-2013.

[14] Martin Fowler. *Patterns of Enterprise Application Architecture*. Reading, Massachusetts: Addison Wesley, Nov. 2002. ISBN: 0321127420.

[15] Erich Gamma et al. *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

[16] Nicolas Haderer, Foutse Khomh and Giuliano Antoniol. 'SQUANER: A framework for monitoring the quality of software systems'. In: *Proceedings of the 2010 IEEE International Conference on Software Maintenance*. ICSM '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–4. ISBN: 978-1-4244-8630-4.

[17] Marty Hall, Larry Brown and Yaakov Chaikin. *Core Servlets and JavaServer Pages, Volume 2: Advanced Technologies*. Prentice Hall, 2007.

[18] M. H. Halstead. *Elements of Software Science*. Elsevier, New York, 1977.

[19] Brian Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice-Hall Object-Oriented Series. Prentice Hall, 1996. ISBN: 9780132398725.

[20] ISO/IEC. *ISO/IEC TR 9126-3. Software engineering - Product quality - Part 3: Internal metrics*. Tech. rep. July 2003.

[21] *JHawk 5.1 Documantation - Metric Guide*. 1.1. Virtual Machinery. Dec. 2012.

[22] Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. Addison Wesley, 2002. ISBN: 0-201-72915-6.

[23] Barbara A. Kitchenham. 'Software quality assurance'. In: *Microprocess. Microsyst.* 13.6 (July 1989), pp. 373–381. ISSN: 0141-9331.

[24] B. Kitchenham and S.L. Pfleeger. 'Software quality: the elusive target [special issues section]'. In: *Software, IEEE* 13.1 (1996), pp. 12–21. ISSN: 0740-7459.

[25] Jean-Louis Letouzey and Thierry Coq. 'The SQALE Analysis Model: An Analysis Model Compliant with the Representation Condition for Assessing the Quality of Software Source Code'. In: *Proceedings of the 2010 Second International Conference on Advances in System Testing and Validation Lifecycle*. VALID '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 43–48. ISBN: 978-0-7695-4146-4.

[26] Virtual Machinery. *JHawk - the Java metrics tool*. http://www.virtualmachinery.com/jhawkprod.htm. Online; accessed 26-Mai-2013.

[27] Virtual Machinery. *Object-Oriented Software Metrics - Class Level Metrics*. http://www.virtualmachinery.com/jhawkmetricsclass.htm. Online; accessed 04-Juni-2013.

[28] Virtual Machinery. *Object-Oriented Software Metrics - Maintainability Index*. http://www.virtualmachinery.com/sidebar4.htm. Online; accessed 04-Juni-2013.

[29] Virtual Machinery. *Object-Oriented Software Metrics - Method Level Metrics*. http://www.virtualmachinery.com/jhawkmetricsmethod.htm. Online; accessed 04-Juni-2013.

[30] T.J. McCabe. 'A Complexity Measure'. In: *IEEE Transactions on Software Engineering* 2.4 (1976), pp. 308–320. ISSN: 0098-5589.

[31] T. J. McCabe and C. W. Butler. 'Design Complexity and Measurement and Testing'. In: *Communications of the ACM* 32(12) (December 1989), pp. 1415–1425.

[32] Jim A McCall, Paul K Richards and Gene F Walters. *Factors in software quality. volume i. concepts and definitions of software quality*. Tech. rep. DTIC Document, 1977.

[33] Sandro Morasca. 'Software Measurement'. In: *Handbook of Software Engineering and Knowledge Engineering. Volume 1: Fundamentals*. Ed. by S. K. Chang. World Scientific Publishing Co., 2001, pp. 239–276. ISBN: 981-02- 4973-X.

[34] Sagar Naik and Piyu Tripathy. *Software testing and quality assurance: theory and practice*. Wiley-Spektrum, 2011.

[35] Paul Oman and Jack Hagemeister. 'Metrics for Assessing a Software System's Maintainability'. In: *Proceedings of the International Conference on Software Maintenance 1992*. IEEE Computer Society Press, Nov. 1992, pp. 337–344.

[36] P Oman, J Hagemeister and D Ash. 'A definition and taxonomy for software maintainability'. In: *Moscow, ID, USA, Tech. Rep* (1992), pp. 91–08.

[37] Oracle. *Java EE*. http://www.oracle.com/technetwork/java/javaee/overview/index.html. Online; accessed 25-Julli-2013.

[38] Oracle. *Msql*. http://www.mysql.com/. Online; accessed 25-Julli-2013.

[39] Squale Project. *SQUALE Software QUALity Enhancement!* http://www.squale.org/. Online; accessed 05-Mai-2013.

[40] Pivotal Software. *Spring Tools*. http://spring.io/tools. Online; accessed 15-December-2013.

[41] SonarSource. *Continuous Code Quality Management*. http://www.sonarsource.com. Online; accessed 26-Mai-2013.

[42] SonarSource. *Sonar*. http://www.sonarsource.org. Online; accessed 26-Mai-2013.

[43] SQALE. *SQALE Software Quality Assessment based on Lifecycle Expectations*. http://www.sqale.org/. Online; accessed 05-Mai-2013.

[44] Kurt D. Welker, Paul W. Oman and Gerald G. Atkinson. 'Development and application of an automated source code maintainability index'. In: *Journal of Software Maintenance* 9.3 (May 1997), pp. 127–159. ISSN: 1040-550X.

[45] Wikipedia. *Hibernate (Java)*. http://en.wikipedia.org/wiki/Hibernate_(Java). Online; accessed 25-Julli-2013.

[46] Wikipedia. *Multitier architecture*. https://en.wikipedia.org/wiki/Multitier_architecture. Online; accessed 25-Julli-2013.